eSNACC 1.7: A High Performance ASN.1 to C/C++ Compiler

Application Programming Interface

Version 1.7

21 April 2004

141 National Business Parkway, Suite 210
Annapolis Junction, MD 20701

<http://www.DigitalNet.com/>

(Originally developed by Michael Sample 1993; msample@cs.ubc.ca
Department of Computer Science, University of British Columbia
6356 Agricultural Rd., Vancouver, British Columbia Canada, V6T 1Z2
AND augmented by: Robert Joop, rj@rainbow.in-berlin.de)

PROLOGUE

This version of SNACC (called eSNACC) is provided by DigitalNet. DigitalNet has made and continues to make many improvements to the original SNACC release by Michael Sample. This manual describes the present version of eSNACC; both the compiler and C/C++ run-time library use.

The Enhanced SNACC ASN.1 library is totally unencumbered as stated in the Enhanced SNACC Software Public License (http://www.digitalnet.com/knowledge/library/snacc/snacc_license.txt) . All source code for the Enhanced SNACC software is being provided at no cost and with no financial limitations regarding its use and distribution. Organizations can use the Enhanced SNACC software without paying any royalties or licensing fees.

ORIGINAL SNACC 1.2rj PROLOGUE

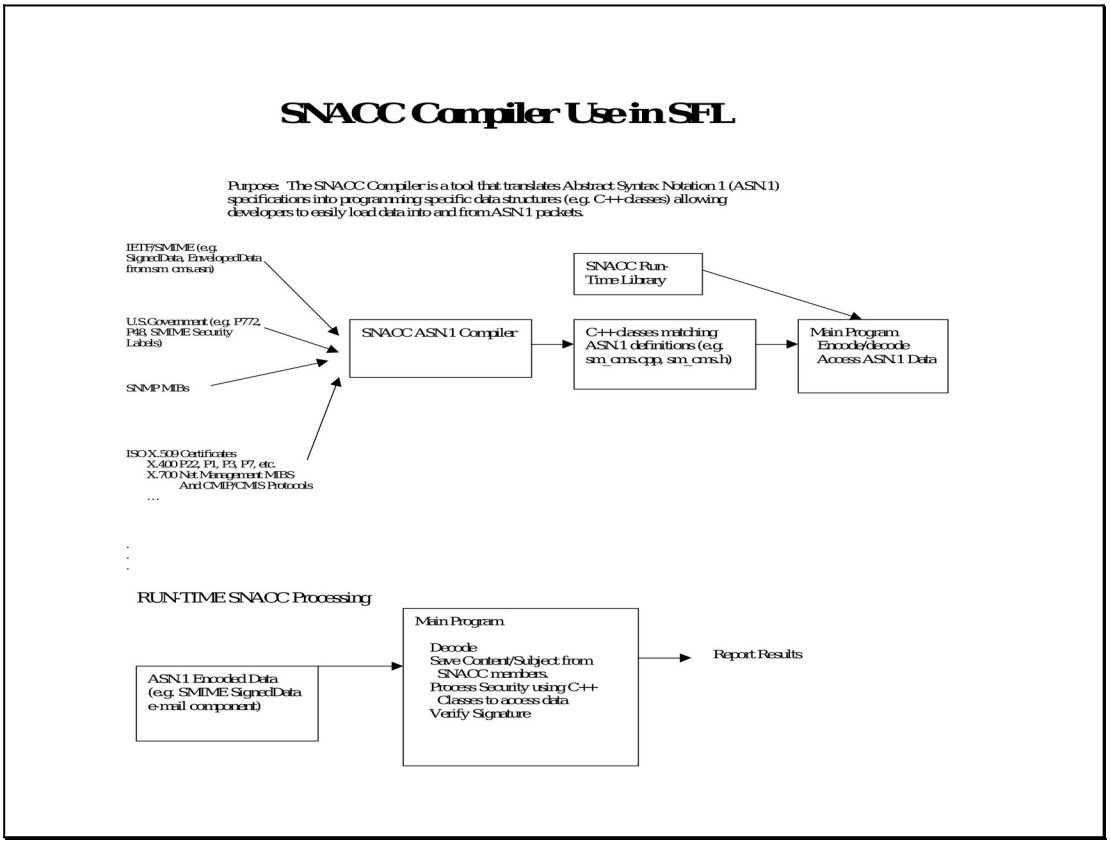
This work was made possible by grants from the Canadian Institute for Telecommunications Research (CITR) and Natural Sciences and Engineering Research Council of Canada (NSERC).

Contents

1 Introduction

eSACC compiles ASN.1 [1] (Abstract Syntax Notation One) modules into C, C++ source code. The generated C or C++ code contains equivalent data structures and routines to convert values between the internal (C or C++) representation and the corresponding BER [2] (Basic Encoding Rules) and DER (Distinguished Encoding Rules) format. The name “snacc” is an acronym for “Sample Neufeld ASN.1 to C/C++ Compiler”.

This compiler basically works on 1990 the ASN.1 syntax. Features have been added to accommodate some recent syntaxes, these are described below. The compiler and C++ run-time library have been rigorously tested on several ASN.1 module suites. Specifically, the SMIME Freeware Library uses the X.509 and IETF CMS specification(s) ASN.1 syntax, compiled by eSNACC. Also, the SNMP V1 test suite for ASN.1 syntax handling of input/decode data (over 10,000 tests for successful decoding and 10,000 tests for graceful error handling; eSNACC handles the entire test suite; see the .tgz. files under ./SNACC/c++-examples/ snmpv1_tests and the source file under ./SNACC/c++-examples/src/snmp.cpp).



The above figure, “SNACC Compiler Use in SFL”, demonstrates the eSNACC compiler use. First, the .cpp and .h source files are built from an ASN.1 specificaion (file) using the eSNACC compiler, then the main application is built using these support files and the eSNACC run-time library. The application program can encode/decode ASN.1 data and reference individual elements using the eSNACC run-time library classes described in this manual.

1.1 DigitalNet/Getronics/J.G.Van Dyke & Associates Update Notes (2004)

These notes refer to the updates performed by DigitalNet/Getronics/J.G.Van Dyke & Associates (all the same company personnel). Improvements have been made to the compiler, run-time libraries, buffer handling, etc. to keep up with recent ASN.1 syntax issues. We continue to develop new features (please check the web site for the most recent release, www.digitalnet.com).

The IDL compiler and TCL support have been left intact, but untested in our release(s). Please inform us of any changes you may make to get these features working; we will update the baseline accordingly. We have no plans to support these features directly.

This version has changed the name from eSNACC to eSNACC to reflect the magnitude of the various updates, summarized below:

- Created MS Windows Visual Studio (6 and .net) projects and workspaces to build on MS Windows as a .dll file. The produced C++ sources allow the user to specify local, IMPORT or EXPORT of symbols from an application .dll executable (see command line options).
- DER (Distinguished Encoding Rules) implemented; BER is no longer used when encoding; decoding will decode both BER and DER data.
- PER (Packed Encoding Rules) implemented; limited support of PER visible constraints.
- Useful ASN.1 types were added as run-time library classes for direct support; this allows for data restrictions.
- Some features from newer ASN.1 specifications have been added; the basic syntax supported is still 1990.
- Makefile(s) were updated; Linux and Solaris are supported directly.
- All IMPORT definitions do not need to be defined in ASN.1 modules to be compiled; this feature allows the user to

specify an include directory containing .asn1 files that will be searched to satisfy any IMPORT definitions of compiled files (very convenient). This allows users to build individual .asn1 modules, not all necessary modules at the same time.

C++ Run-time library updates follow:

- AsnInt now handles big integers, greater than 4 bytes (signed and unsigned)
- AsnBits enchanced to construct BitStrings from binary strings directly.
- Added AsnSetOf and AsnSeqOf templates.
- All string classes are directly supported, not through the useful types ASN.1 definitions. These classes check that data encoded/decoded is valid for the appropriate string type.
- Updated Exception handling (see snaccexcept.h)
- AsnAny table/processing updates; much more flexible. Any(s) can now be treated just like any other AsnType for encode/decode operations, no need for custom code for simple encode/decode operations.
- Added C++ namespace features for unique symbol references.
- Consolidated include files of individual eSNACC C++ run-time classes into a single .h file for ease of use.
- XML printing was added to the run-time library and to the compiler generated classes for convenient display of binary ASN.1 results.
- All references to AsnList are now std::list
- Extensibility is now supported for the Set/Sequence/Choice syntaxes
- Relative-Oid type is supported
- No longer supports un-named types (2002 syntax compliant)

“C” Run-time library updates follow:

- String encode/decode operations now check that the string information is valid. (ASN.1 String definitions are now native to the eSNACC compiler for “C” as well).
- AsnAny processing for “C” has been improved, no custom code is necessary for simple encode/decode operations.

- Relative-Oid type is supported

1.2 MS Windows Building eSNACC

In MS Visual Studio, load the "./SNACC/snacc_builds.dsw" (or "snacc_builds.sln" for .net) and execute the "buildall" project. This will build the compiler, run-time libraries for "C" and C++, and build the test application.

1.3 Unix/Linux Configuring and Installing eSNACC

The build process has changed considerably from the previous release. Now all you have to do to build on a clean distribution is:

configure

make

or

make debug

From the top of the source tree (./SNACC). If you want to clean the source tree of all objects and libraries:

make clean

If you are going to reuse the same source tree on another platform make sure you

remove all platform specific files by doing:

make distclean

`make` (GNU `make` is recommended)

Some versions of yacc may choke due to the large size of the parse-asn1.y file, however, we have had no problems with bison . Our yacc grammar for ASN.1 has 61 shift/reduce errors and 2 reduce/reduce errors. Most of these errors were introduced when certain macros were added to the compiler. Some of the shift/reduce errors will require you to follow the offending macro in the ASN.1 module with a semi-colon. The reduce/reduce errors were introduced by macros that have "Type or Value Lists" because the NULL Type and NULL values use the same symbol, "NULL". This is not a problem since no real processing is done with the macros in question at the present.

Lex will work for the lex-asn1.l file but *flex* will typically produce a smaller executable. Most versions of lex have a small maximum token size that will cause problems for long tokens in the ASN.1 source files, such as quoted strings. To avoid this problem,

increase the YYLMAX value in the generated lex-asn1.c file to at least 2048. Flex does not seem to have this problem.

The configuration process has been simplified (at least for the installer of eSNACC) by the use of GNU autoconf.

The only file that may have to be edited is .../policy. h. It contains a few compilation switches you may want to toggle.

The eSNACC compiler and library C code has been written to support ANSI or non-ANSI C. The configuration script tries to find out whether your C compiler understands ANSI C.

The configuration script generates two files:

.../makehead gets included by all makefiles. It contains a lot of definitions used by make.

.../config.h contains all the machine, operating system, compiler and environment dependent settings. It is included by .../snacc.h.

If you wish to install only the C (including type tables) or only the C++ versions of the library, type make c or make c++, respectively, instead of make. If the make succeeds, the eSNACC binary should be present as .../compiler/snacc , the C runtime libraries, libasn1csbuf.a , libasn1cebuf.a , libasn1cmbuf.a and libasn1ctbl.a , should be in .../c-lib/ and the C++ runtime library, libasn1c++.a (and, if you compiled with the Tcl option enabled, another run-time library, libasn1tcl.a), should be in .../c++-lib/. The type table tools, ptbl , pval and mkchdr, will be in their respective directories under .../ tbl-tools/ .

To avoid warnings across platforms, you must run lex/flex and bison/yacc on their respective platforms.

To install eSNACC, you can call “make install”. This installs the eSNACC compiler binary, the libraries, the .h and .asn1 files, the type table tools, as well as the manual pages into the usual directories (/usr/local on Unix based platforms, %windir%/system32 for MS Windows).

1.4 Running eSNACC

eSNACC is typically invoked from the shell command line. With no arguments, the “usage” report is printed:

```
Usage: C:\devel.d\develCurent.d\SMPDist\bin\esnaccd.exe [-h] [-P] [-t] [-v]
[-e]
 [-d] [-p] [-f] [-a] [-b]
        [-c | -C [cpp] | -T <table output file> | -idl ]
        [-mm] [-mf <max file name length>]
```

```
                [-l <neg number>]
                [-VDAexport=DEFINE_NAME] to designate export of SNACC
generated clas
        ses
                [-E BER|DER select encoding rules to generate (C only)]
                [-D NECESSARY for VDA Rules (ANY processing)
                <ASN.1 file list>


        -h   prints this msg
        -c   generate C encoders and decoders (default)
        -C   generate C++ encoders and decoders
        -novolat   for broken C++ compilers: return *this after calling abort()
        -T <filename> write a type table file for the ASN.1 modules to file
filename
        -O <filename> writes the type table file in the original (<1.3b2) format
        -idl generate CORBA IDL
            (i.e. PrintableString). See the useful.asn1 file (in the
            snacc/asn1specs/ directory).
        -P   print the parsed ASN.1 modules to stdout from their parse trees
            (helpful debugging)
        -t   generate type definitions
        -v   generate value definitions (limited)
        -e   generate encode routines
        -d   generate decode routines
        -p   generate print routines
        -f   generate hierarchical free routines (C only)
           note: if none of -t -v -e -d -p -f are given, all are generated.
           These do not affect type tables.
        -mm  mangle output file name into module name (by default, the output
file
            inherits the input file's name, with only the suffix replaced)
        -mf <num> num is maximum file name length for the generated source
files
        -l <neg num> where to start error longjmp values decending from
(obscure).
        -I <Directory Path> ASN.1 directory path for supporting ASN.1
modules.
        -b  generates PER encode/decode routines
```

Use `-' as the ASN.1 source file name to parse stdin. (ONLY FOR Unix platforms)

This ASN.1 compiler produces C or C++ BER encoders and decoders or type tables.

Version 1.5

Release Date: 2003-02-20

Please see imc-snacc@imc.org for new versions and where to send bug reports and

comments.

eSNACC generates C or C++ source code for BER encode and decode routines as well as print and free routines for each type in the given ASN.1 modules. Alternatively, eSNACC can produce type tables that can be used for table based/interpreted encoding and decoding. The type table based methods tend to be slower than their C or C++ counterparts but they tend to use less memory (table size vs. C/C++ object code).

eSNACC may also be used to generate CORBA IDL. This part of eSNACC is very new and I would rate it as pre-alpha. (This feature has not been kept up-to-date.)

The `-meta` , `-mA` , `-mC` and `-tcl` options are only present when the Tcl and Tk libraries were found at configuration time. (This feature has not been kept up-to-date.)

Most of the 1990 ASN.1 features are parsed although some do not affect the generated code. Fairly rigorous error checking is performed on the ASN.1 source; any errors detected will be reported (printed to *stderr*.)

Each file in the ASN.1 file list should contain a complete ASN.1 module. ASN.1 modules that use the IMPORTS feature must reference the other ASN.1 modules on the command line (specify all necessary modules in the ASN.1 file list OR indicate the include directory to extract the information from, "-I"). The generated source files will include each module's header file in the command line order. This makes it important to order the modules from least dependent to most dependent on the command line to avoid type ordering problems. Currently, snacc assumes that each ASN.1 file given on the command line depends on all of the others on the command line. Only the header files from modules referenced in the import list for that module are included.

If the target language is C, eSNACC will generate a `.h` and `.c` file for each specified ASN.1 module. If the target language is C++, eSNACC will generate a `.h` and `.cpp` file for each module. If the target language is CORBA IDL, eSNACC will generate an `.idl` file for each module. The generated file names will be derived from the module's filenames, or from the module names if the `-mm` command line switch has been given.

The command line options are:

-I<Directory Path> to specify an include directory for .asn1 modules referencing all IMPORT variables. This feature allows the user to build individual .asn1 modules, not have all necessary definitions built at the same time.

-h short for "help", prints a synopsis of eSNACC and exits.

-c causes eSNACC to generate C source code. This is the default behavior of eSNACC if neither of the `-c` or `-C` options are given. Only one of the `-c`, `-C`, `-idl` or `-T` options should be specified.

-C causes eSNACC to generate C++ source code.

-idl causes eSNACC to generate CORBA IDL source code.

-T *file* causes eSNACC to generate type tables and write them to the given file *file*.

-meta *types* causes eSNACC to generate C++ classes with type meta information. Requires C++ functionality and therefore implies `-C` (C++ code generation). The *types* denote the PDUs and have the following syntax: a comma-separated list of pairs of:

module name, a dot, and a type name from that module. (Example: `snacc -tcl M1. T-a, M-2. Tb mod1.asn1 m2.asn1`)

–mA and **–mC** causes the metacode to use identifiers as defined in the ASN.1 source files or as used in the generated C++ code, respectively. (It defaults to `-mC`.)

–tcl *types* causes eSNACC to generate functions for a Tcl interface. Needs the type meta information and thus implies `-meta` (see above). The `-meta` option can and should be omitted, the *types* are as for the `-meta` option (the *types* arguments are additive, in case you specify both options).

–P causes eSNACC to print the parsed ASN.1 modules to *stdout* after the types have been linked, sorted, and processed. This option is useful for debugging eSNACC and observing the modifications eSNACC performs on the types to make code generation simpler.

The options, `-t, -v, -e, -d, -p,` and `-f` affect what types and routines go into the generated source code. These options do not affect type table generation. If none of them are given on the command line, eSNACC assumes that all of them are in effect. For example, if you do not need the Free or Print routines, you should give the `-t -v -e -d` options to eSNACC. This lets you trim the size of the generated code by removing unnecessary routines; the code generated from large ASN.1 specifications can produce very large binaries.

–t causes eSNACC to generate type definitions in the target language for each ASN.1 type.

–v causes eSNACC to generate value definitions in the target language for each ASN.1 value. Currently value definitions are limited to INTEGERs, BOOLEANs and OBJECT IDENTIFIERs.

–e causes eSNACC to generate encode routines in the target language for each ASN.1 type.

–d causes eSNACC to generate decode routines in the target language for each ASN.1 type.

–p causes eSNACC to generate print routines in the target language for each ASN.1 type.

–f causes eSNACC to generate free routines in the target language for each ASN.1 type. This option only works when the target language is C. The free routines hierarchically free C values. A more efficient approach is to use the provided nibble-memory system. The nibble memory permits freeing an entire decoded value without traversing the decoded value. This is the default memory allocator used by eSNACC generated decoders. See file

`.../c-lib/inc/asn-config.h` to change the default memory system. For more information on the memory management see Section 5.14.

–u *file* causes eSNACC to read the useful types definitions from the ASN.1 module in file *file* for linking purposes. For some ASN.1 specifications, such as SNMP, the useful types are not needed. The types in the given useful types file are globally available to all modules; a useful type definition is overridden by a local or explicitly imported type with the same name. The original useful type module that defined the individual string elements is no longer necessary since all string types were incorporated into the compiler and “C”/C++ run-time libraries.

–mm This switch is supplied for backwards compatibility. eSNACC versions 1.0 and 1.1 produced files with names generated from the ASN.1 module name contained in the input file. Snacc 1.2rj by default retains the input file name and replaces the suffix only. The new behavior makes makefile writing easier, as with modern makes, pattern matching can be used.

–mf number causes the names of the generated source files to have a maximum length of number characters, including their suffix. The number argument must be at least 3. This option is useful for supporting operating systems that only support short file names. A better solution is to shorten the module name of each ASN.1 module.

–l number this is fairly obscure but may be useful. Each error that the decoders can report is given an id number. The number *number* is where the error ids start decreasing from as they are assigned to errors. The default is -100 if this option is not given. Avoid using a number in the range -100 to 0 since they may conflict with the library routines' error ids. If you are re-compiling the useful types for the library use -50. Another use of this option is to integrate newly generated code with older code; if done correctly, the error ids will not conflict.

Since ASN. 1 has different scoping rules than C and C++, some name munging is done for types, named-numbers etc. to eliminate conflicts. Some capitalization schemes were chosen to fit common C programming style. For all names, dashes in the ASN.1 source are converted to underscores. See Sections 4.2 and 6.2 for more naming information.

If the -mm switch has been given, the module name is used as a base name for the generated source file names. It will be put into lowercase and dashes will be replaced with underscores. Module names that result in file names longer than specified with the -mf

option will be truncated. If the -mf option was not given, file names will be truncated if they are too long for the target file system. You may want to shorten long module names to meaningful abbreviations. This will avoid file name conflicts for module names that are truncated to the same substring. Any module name and file name conflicts will be reported.

If your ASN.1 modules have syntactic or semantic errors, each error will be printed to *stderr* along with the file name and line number of where it occurred. These errors are usable by GNU emacs compiling tools. See the next chapter for more information on the types of errors eSNACC can detect.

More errors can be detected and reported in a single compile if type and value definitions are separated by semi-colons. Separating type and value definitions with semi-colons is not required, and if used, need not be used to separate all type and value definitions. Semi-colons are necessary after some macros that introduce ambiguity. In general, if you get a parse error you can't figure out, try separating the surrounding type/value definitions with semicolons.

1.4.1 Known Bugs

- eSNACC has problems with the following case:

  ```
  Foo::=SEQUENCE
  {
    id IdType,
    val ANY DEFINED BY id
  }
  IdType::=CHOICE
  {
    a INTEGER,
    b OBJECT IDENTIFIER
  }
  ```

- The error checking pass will print an error to the effect that the id type must be INTEGER or OBJECT IDENTIFER. To fix this you must modify the error checking pass as well as the code generation pass. To be cheap about it, disable/fix the error checking and hand modify the generated code.
- The hashing code used for handling ANY DEFINED BY id to type mappings will encounter problems if the hash table goes

more than four levels deep (I think this is unlikely). To fix this just add linear chaining at fourth level.

- The `.../configure` script should check whether the machine's floating point format is IEEE or whether the IEEE library exists.
- The C++ library severely lacks a convenient buffer management class that automatically expands like the C libraries' ExpBuf. What use is an efficient buffer management when you have got to build a loop around eSNACC's encoding routine that reallocates larger buffers until the result fits?
- Where this document describes personal experiences, it is usually unclear to which author `I' refers. (One way to find out is to look at eSNACC 1.1' s docu- mentation.)

1.5 Reporting Bugs and Your Own Improvements

eSNACC is actively supported by DigitalNet, please send any suggestions, bug-fixes, improvements, etc. to Robert.Colestock@DigitalNet.com (or check the web site under knowledge bank, www.digitalnet.com, for recent eSNACC issues/contact information).

1.6 Version Updates

New in 1.7

- Enchancements to C++ runtime:
  - Support of constraints checking for BER/PER
  - Added Asn-Relative-Oid's
  - Updated Asn-Oid to be inherited from Asn-Relative-Oid
  - Added Extensibility to the set/sequence/choice syntax (BER encoding/decoding only, PER is not yet supported)
  - asn::list has been changed to std::list (many changes)
  - non-optional set/sequence/choice elements are no longer generated as pointers
  - PER encode/decode capability for both aligned and unaligned variants (see PER beta notes below)
- Enchancements to C runtime:
  - Added useful types
  - Added Asn-Relative-Oid's

- Enhancements to compiler
- eSNACC no longer supports un-named types (2002 syntax update)
- added -b compiler option to turn on/off PER encoding/decoding function generation in set/sequence/choice (Note -- calling PEnc/PDec will still work, but will not produce correct encoding unless -b is used!!)

New in EKMS PER Beta (Packed Encoding Rules)

- Aligned and unaligned PER variants (C++ only)
- Limited constraint checking and PER encoding for PER visible constraints
  - Supported
    - char Stringtypes
    - Integer
    - Octet String
    - Bit String
    - Sequence-of / Set-of (limited)
  - Currently Unsupported
    - wide char stringtypes
    - extensibility in constraints

New in 1.6Beta

- Updated "C" library to automatically handle ANY load/unloads as buffers.
- Added interpretation of ASN.1 integer constants as values in tag references for "C" and C++.
- Added "--snacc namespace: " pre-processor feature for unique C++ ASn.1
- module namespace references.
- Updated SNACC document (in the ./SNACC/doc directory) to present DigitalNet
- updates/enhancements.
- Updated c++-examples and c-examples to demontrate recent features.

New in 1.5

- Updated "C" library to automatically handle ANY load/unloads as buffers.
- Added interpretation of ASN.1 integer constants as values in tag references for "C" and C++.
- Added "--snacc namespace: " pre-processor feature for unique C++ ASn.1 module namespace references.
- Updated SNACC document (in the ./SNACC/doc directory) to present DigitalNet updates/enhancements.
- Updated c++-examples and c-examples to demontrate recent features.

New in 1.4

- rewrote makefiles to make build process easier and faster.
- Enchancements to C++ runtime:
- AsnInt changed to be so that it no longer inherits AsnOcts
- AsnBits enchanced to construct BitStrings from binary strings.
- Added AsnSetOf and AsnSeqOf templates.
- Added Exception handling (see snaccexcept.h)
- Moved BDecPdu to AsnType. So every type as access to it now. This was done to help reduce the number of symbols & methods the compiler generates.
- Added useful types
- Enchancements to C runtime:
- Added useful types
- Enhancements to compiler
- Removed -u switch because useful types are now in the runtime library.
- Added useful types as basic types.

2 C Code Generation

2.1 Introduction

eSNACC was designed primarily to provide high-performance encoders and decoders. Key areas to optimize are buffer and memory management. Buffers are used to hold encoded values and the memory management is used when building the internal representation of a value when decoding.

C macros are used where possible to eliminate function call overhead for small, commonly used routines. Using macros with constant expressions as parameters allows smarter C compilers to do some of the calculations at compile time. In general, short-cuts that can be taken without sacrificing the robustness of code are used.

The generated code can be quite large; large reductions of the size of the binaries can be achieved by using the optimizing options of your C compiler.

We will use an example ASN.1 module, EX1, to help explain eSNACC's code generation. The EX1 module uses some of the common built-in types and contains some simple values. The field names have been left out to show eSNACC naming conventions. The C generation code is in `.../compiler/back-ends/c-gen/` if you want to alter it.

```
EX1 DEFINITIONS ::=
BEGIN
anOidVal OBJECT IDENTIFIER ::= { joint-iso-ccitt 40 foobar( 29) }
theSameOidVal OBJECT IDENTIFIER ::= { 2 40 29 }
anIntVal INTEGER ::= 1
aBoolVal BOOLEAN ::= TRUE
T1 ::= SEQUENCE
{
        INTEGER OPTIONAL,
       OCTET STRING OPTIONAL,
       ENUMERATED { a( 0), b( 1), c( 2) },
        SEQUENCE OF INTEGER,
       SEQUENCE { id OBJECT IDENTIFIER, value OCTET STRING },
       CHOICE { INTEGER, OBJECT IDENTIFIER }
} END
```

Use the following command to compile the EX1 ASN.1 module:

```
%1 snacc -u .../asn1specs/asn-useful.asn1 .../asn1specs/ex1.asn1
```

This produces the files `ex1.h` and `ex1.c`.

For each ASN.1 type an equivalent C data type, a DER encoding routine, a BER/DER decoding routine, a printing routine and a freeing routine will be generated. C values will also be generated from simple ASN.1 values. Each aspect of the C code generation will be discussed in the next sections.

2.2 ASN.1 to C Naming Conventions

For any given module, eSNACC may produce C type definitions, functions and *#defines*. We assume that all C *typedef, struct, enum* and *union* tag, *enum* value, variable, *#define* and function names share a single name space.

The C type name for a type is the same as its ASN.1 type name (with any hyphens converted to underscores) unless there is a conflict. Since, unlike ASN.1, the C types for each ASN.1 module share the same name space, eSNACC makes sure the C typenames are unique among all the modules and that they do not conflict with C keywords. The conflicts are resolved by appending digits to the conflicting name. To avoid confusing numbered type names etc., you should edit the ASN.1 source and name them properly.

Named numbers, ENUMERATED values and named bits are put in entirely in upper case to match the common C convention for *#define* and *enum* values.

Empty field names in SETs, SEQUENCEs, and CHOICEs will be filled. The field name is derived from the type name for that field. The library types such as INTEGER have default field names defined by the compiler (see `.../compiler/back-ends/c-gen/rules.c`). The first letter of the field name is in lower case. Again, empty field names should be fixed properly by adding them to the ASN.1 source.

New type definitions will be generated for SETs, SEQUENCEs, CHOICEs, ENUMERATED, INTEGERs with named numbers and BIT STRING with named bits whose definitions are embedded in other SET, SEQUENCE, SET OF, SEQUENCE OF, or CHOICE definitions. The name of the new type is derived from the name of the type in which it was embedded. Perhaps a better way would use the field name as well, if present.

2.3 ASN.1 to C Data Structure Translation

To handle the different scoping rules between ASN.1 and C, the names of some ASN.1 data structure elements such as ENUMERATED type symbols may be altered to avoid conflicts. The T1 type in example ASN.1 module EX1 has no field names so eSNACC will generate them. It is recommended to provide field names in the ASN.1 source instead of relying on compiler generated names. The following is the generated C data structure for the EX1 module from the `ex1.h` file (function prototypes have been removed):

```
typedef enum
{
      A = 0,
      B = 1,
       C = 2
} T1Enum; /* ENUMERATED { A( 0), B( 1), C( 2) } */
typedef struct T1Choice /* CHOICE */
{
      enum T1ChoiceChoiceId
      {
            T1CHOICE_ INT1,
            T1CHOICE_ OID
      } choiceId;
      union T1ChoiceChoiceUnion
      {
            AsnInt int1; /* INTEGER */
            AsnOid *oid; /* OBJECT IDENTIFIER */
      } a;
} T1Choice;
typedef struct T1Seq /* SEQUENCE */
{
      AsnOid id; /* OBJECT IDENTIFIER */
      AsnOcts value; /* OCTET STRING */
} T1Seq;
typedef AsnList T1SeqOf; /* SEQUENCE OF INTEGER */
typedef struct T1 /* SEQUENCE */
{
      AsnInt *int1; /* INTEGER OPTIONAL */
```

```
        AsnOcts octs; /* OCTET STRING OPTIONAL */
        T1Enum t1Enum; /* T1Enum */
        T1SeqOf *t1SeqOf; /* T1SeqOf */
        struct T1Seq *t1Seq; /* T1Seq */
        struct T1Choice *t1Choice; /* T1Choice */
} T1;
```

Every ASN.1 type definition maps into a C *typedef*. SETs and SEQUENCEs map into C structures and other simple types map into their obvious C counterpart. SET OF and SEQUENCE OF types map into a generic list type which is doubly linked and NULL terminated. The reverse link on the lists allows for simpler backwards encoding. More information on the library types can be found in Chapter 5.

Comments that contain a fragment of each type's ASN.1 definition are inserted in the header file to clarify cases where elements have been renamed.

Aggregate types that are defined in other type definitions are moved to their own type definitions. For example, notice how the SEQUENCE and CHOICE that are in type *T1* have been moved to the types *T1Seq* and *T1Choice* in the C code. This simplifies code generation at the cost of introducing new types.

Identifiers for named numbers from INTEGER and ENUMERATED types and named bits from the BIT STRING type are capitalized in the C representation. The ENUMERATED type maps to a C *enum* and the INTEGER and BIT STRING named numbers/ bits are handled with *#define* statements.

Most OPTIONAL elements of SEQUENCEs and SETs are referenced by pointer. An element is considered present if its pointer is non-NULL. OCTET STRINGs, BIT STRINGs and OBJECT IDENTIFIERs are the exceptions, and are included by value even when they are OPTIONAL because they are small and contain an internal pointer that can be used to determine their presence. For an example of this, look at the first two elements of type *T1*. The INTEGER type is referenced by pointer because it is OPTIONAL, but the OCTET STRING type is included (non-pointer) in the *T1* type even though it is OPTIONAL.

2.4 Encode Routines

eSNACC generates two kinds of encoding routines. One is PDU oriented and encodes the type's tag, length and content and the other only encodes the type's content. The generated encoders only call the content encoders, except in the case of ANY and ANY

DEFINED BY types. Typically, you will only call the PDU oriented routines from your code.

The content and PDU encoding routine interfaces are similar for all ASN.1 types. They both take two parameters, one is a buffer pointer and the other is a pointer to the value to be encoded. For example the *T1* type from module EX1 has the following prototypes for its encoding routines.

```
AsnLen BEncT1Content (BUF_ TYPE b, T1 *v);

AsnLen BEncT1 (BUF_ TYPE b, T1 *v);
```

BEnc is short for “BER Encode”. The BUF_*TYPE* parameter is the buffer to encode the value into and the *T1 ** parameter is a pointer to the instance of the *T1* type that is to be encoded.

The *BEncT1Content* routine only encodes the content of a *T1* type and returns its encoded length; it does not encode its tag (UNIVERSAL (CONSTRUCTED) 16 for SEQUENCE) or length. The job of encoding the tag and length is up to any type that encapsulates *T1*. This design allows decisions about implicit tagging to be made at code generation time instead of runtime, improving performance. Also, different encoding rules may fit into this model more easily.

The *BEncT1* routine encodes the tag (UNIVERSAL (CONSTRUCTED) 16 for SE- QUENCE), length and content of a *T1* type and returns its encoded length. This is the PDU oriented routine and will only be generated if the user designates the type as a PDU type via a compiler directive or the type is used as the content of an ANY or ANY DEFINED BY type (as indicated by an OBJECT-TYPE macro). A PDU type is a type that defines an entire PDU; the user will typically be calling the encode and decode routine for PDU types directly. See Section 4.9 for how to designate PDU types with compiler directives.

The eSNACC encoders are somewhat strange; they encode a value starting from the end of its BER representation and work back to its beginning. This “backwards” encoding technique simplifies the use of definite lengths on constructed values. Other encoders that encode forwards, such as those of CASN1, use an intermediate buffer format so that a buffer containing the encoded length of a constructed value can be inserted before its encoded content, after the content has been encoded. Use of intermediate buffers hurts performance. Other compilers' approaches have been to only encode indefinite lengths for constructed values, however, this will not support some encoding rules such as DER. The drawback of encoding backwards is that BER values cannot be written to stream-oriented connections as they are encoded.

Both definite and indefinite length encodings for constructed values' lengths are sup- ported. Currently the choice is made when compiling the generated code, via the *USE_INDEF_LEN* flag. If both length forms, definite and indefinite, are required, it easy to modify the length encoding macros to check a global variable for the length form to use. For most types, using definite lengths produces smaller encodings with little performance difference.

After calling an encode routine you should always check the buffer you encoded into for a write error. This is the only error reporting mechanism used for the encoders. See the C buffer section (Section 5.13) for how to check a buffer for a write error.

2.5 Decode Routines

Decoding routines are like the encoding routines in that there are two kinds, one that decodes the type's tag, length and content and one that only decodes the type's content. As mentioned in the encoder section, the content style interface allows implicit tagging decisions to be made at compile time.

Unlike the encoding routines, the PDU and content decoding routines take different arguments. For the *T1* type the following would be produced:

```
void BDecT1Content (BUF_ TYPE b, AsnTag tagId0, AsnLen elmtLen0, T1
*v, AsnLen *bytesDecoded, ENV_ TYPE env);

void BDecT1 (BUF_ TYPE b, T1 *v, AsnLen *bytesDecoded, ENV_ TYPE
env);
```

Notice that the content decoder, *BDecT1Content*, has tag and length parameters that the PDU decoder, *BDecT1*, does not have. Since the content decoder does not decode the tag and length on the value, it is necessary to pass them in as parameters. Only OCTET STRING and BIT STRING decoders will actually use the information in the tag parameter.

The *BUF_TYPE* parameter is the buffer that holds the BER value being decoded.

The *tagId0* parameter is the last tag that was decoded on the content of the type that is about to be decoded. In the case of type *T1, BDecT1Content* gets a tagId0 of UNIVERSAL (CONSTRUCTED) 16, unless it is implicitly tagged by another type. Most content decoding routines ignore the tag information. OCTET STRING and BIT STRING decoders use the tag information to determine whether the contents are constructed or primitive. CHOICE decoders use the tag information to determine which CHOICE

element is present. CHOICE values are treated differently, as will be explained shortly.

The *elmtLen0* parameter is the length of the content of the type being decoded. This is simply the length decoded from the buffer by the containing type's decoder just before calling this decode routine.

The *v* parameter is a pointer to space allocated for the type being decoded. This memory is not allocated by the decoding routine itself; this supports the cases where the type is enclosed in the struct of the parent (i.e. no extra allocation is necessary). If the type to be decoded is referenced by pointer from its parent type, the parent type's decoding routine must allocate the type.

The *bytesDecoded* parameter maintains the running total of the number of octets that have been decoded. For example, if I call *BDecT1Content* with a *bytesDecoded* parameter that points to 20 and the encoded length of the *T1* value is 30 octets, *bytesDecoded* will point to 50 when *BDecT1Content* returns. Maintaining the length is vital to determining the presence or absence of OPTIONAL elements in a SET or at the end of SEQUENCE. Local variables are used to hold the lengths; there is no global stack of lengths as with CASN1.

The *env* parameter is used in conjunction with *longjmp*calls. When an decoder encounters a fatal error such as a missing tag, it uses the *env* with a *longjmp*call to pop back to the initial decode call. Section 5.15 has more error management details.

CHOICEs are decoded a little differently from other types. For all types except CHOICEs, all of the tag and length pairs on the content are decoded by the parent type, and the last pair is passed into a content decoding routine via the *tagId0* and *elmtLen0* parameters. For CHOICEs, all of the tag and length pairs on the content are decoded and then the first tag and length pair in the CHOICE content is decoded by the parent and passed into the CHOICE content decoding routine. The first tag in a CHOICE's content is the important tag by which the CHOICE determines which element is present. This technique simplifies the code for dealing with untagged CHOICEs embedded in other CHOICEs. CHOICEs nested in this way mean that a single tag determines which element is present in more than one CHOICE.

The decoding routines allocate memory to hold the decoded value. By default eSNACC decoders use nibble memory (see Section 5.14) which is very efficient in allocation and virtually cost free for freeing.

To save memory, decoders generated by some other tools build values that reference the data in the encoded PDU for types like OCTET STRING. eSNACC decoded values do not reference the BER data in any way for several reasons. One, the encoded value may be held in some bizarre buffer making access to the value difficult. Two, with more encoding rules being formalized, this technique may not always work since the encoded format may be different from the desired internal format. Three, eSNACC decoders concatenate any constructed BIT and OCTET STRINGs values when decoding, to simplify processing in the application.

eSNACC decoders can detect a variety of errors which will be reported by *longjmp*. Any tagging errors are reported. SETs must contain all non-OPTIONAL components and SEQUENCEs must be in order and contain all non-OPTIONAL components. Extra components in SETs and SEQUENCEs are considered an error. Errors will also be reported if you attempt to decode values that exceed the limitations of the internal representation (e.g. an integer that is larger than a *long int* allows).

2.6 Print Routines

All of the generated print routines take similar parameters. For example the *T1* type' s print routine prototype is:

```
void PrintT1 (FILE *f, T1 *v, unsigned short int indent);
```

The print routine writes the given value, v, to the given *FILE *, f*. The printed value is indented by *indent* spaces. The values are printed in an ASN.1 value notation style. *PrintT1* prints in the following style:

```
{ -- SEQUENCE –
   17,
   '436c696d6220617420537175616d697368'H -- "Climb at Squamish" --,
   0,
   {
       -- SEQUENCE OF –
       18,
       19
    },
    { -- SEQUENCE –
       id {2 40 29},
       value '736f6d6520737472696e67'H -- "some string" –
    },
```

```
    20
}
```

OCTET STRINGs are printed in a hexadecimal notation, and any printable characters are included after the string in an ASN.1 comment. Note that the enumerated type value, 0, did not print its symbol, “A” from the ENUMERATED type. It would be fairly easy to modify the C and C++ back ends to generate print routines that printed the ENUMERATED types' symbols instead of their values.

2.7 Free Routines

eSNACC generates free routines of the form:

```
void FreeT1 (T1 *v);
```

These routines will free all the components named type. For example the above *FreeT1* routine will free all the components of the given *T1* value, but not the *T1* value itself. The passed in pointer is not freed because it may be embedded in another type which will be freed by another call to *Asn1Free*. All the pieces of memory are freed using the *Asn1Free* macro defined in `asn-config.h`. Each library type has its own free routine that may call *Asn1Free.* The values are typically allocated during decoding, using the *Asn1Alloc* macro.

The memory management can be changed by editing the `asn-config.h` file to use your own memory management routines. By default the memory manager uses the nibble memory system described in Section 5.14. The nibble memory system does not need explicit frees of each component so the generated free routines are not needed. However, if you change the memory management to use something like *malloc* and *free*, you should use the generated free routines.

2.8 ASN. 1 to C Value Translation

C values will be produced for INTEGER, BOOLEAN and OBJECT IDENTIFIER values. C *extern* declarations for the value are put at the end of the header file (after all of the type definitions). The value definitions are put at the beginning of the source file. For example, the following will be produced for the EX1 module (at the end of file ex1. h):

```
extern AsnOid anOidVal;
extern AsnOid theSameOidVal;
extern AsnInt anIntVal;
extern AsnBool aBoolVal;
```

```
extern AsnInt foobar;
(at the beginning of file ex1.c):
AsnOid anOidVal = { 2, "\ 170\ 35" };
AsnOid theSameOidVal = { 2, "\ 170\ 35" };
AsnInt anIntVal = 1;
AsnBool aBoolVal = TRUE;
AsnInt foobar = 29;
```

2.9 Compiler Directives

eSNACC allows the user to control some aspects of the generated code by inserting special comments in the ASN.1 source. Warning! only the *isPdu*directive has been tested to any extent. Use the others very carefully and only if you really need to. The compiler directives have the form:

```
--snacc <attribute>:"< value>" <attribute>:"< value>" ...
```

The *attribute* is the name of one of the accepted attributes and the *value* is what the *attribute's* new value will be. The attribute value pairs can be listed in a single *–snacc* comment or spread out in a list of consecutive comments.

Compiler directives are only accepted in certain places in the ASN.1 code. Depending on their location in the ASN.1 source, the compiler directives affect type definitions or type references. The directives for type definitions and references are different. Module level compiler directives to specify output file names and other information would be useful, but are not implemented.

Here is an example to present some of the compiler directives and their uses. Let's say your data structure always deals with *PrintableStrings* that are null terminated (internally, not in the encoding). The default eSNACC string type is a structure that includes a length and *char ** for the string octets. To change the default type to a simple *char ** the best way would be define your own string type, let' s say *MyString* as follows:

```
Foo ::= SET
{
        s1 [0] MyString OPTIONAL,
        s2 [1] MyString,
        i1 [2] INTEGER
}
Bar ::= CHOICE
```

```
{
        s1 MyString,
        i1 INTEGER
}
Bell ::= MyString
MyString ::= -- snacc isPtrForTypeDef:" FALSE"
             -- snacc isPtrForTypeRef:" FALSE"
             -- snacc isPtrInChoice:" FALSE"
             -- snacc isPtrForOpt:" FALSE"
             -- snacc optTestRoutineName:" MYSTRING_ NON_ NULL"
             -- snacc genPrintRoutine:" FALSE"
             -- snacc genEncodeRoutine:" FALSE"
             -- snacc genDecodeRoutine:" FALSE"
             -- snacc genFreeRoutine:" FALSE"
             -- snacc printRoutineName:" printMyString"
             -- snacc encodeRoutineName:" EncMyString"
             -- snacc decodeRoutineName:" DecMyString"
             -- snacc freeRoutineName:" FreeMyString"
             PrintableString -- snacc cTypeName:" char *"
```

All but the last *–snacc* comment bind with the *MyString* type definition. The last directive comment binds with the *PrintableString* type. The C data structure resulting from the above ASN.1 and compiler directives is the following:

```
typedef char *MyString; /* PrintableString */
typedef struct Foo /* SET */
{
        MyString s1; /* [0] MyString OPTIONAL */
        MyString s2; /* [1] MyString */
        AsnInt i1; /* [2] INTEGER */
} Foo;
typedef struct Bar /* CHOICE */
{
   enum BarChoiceId
   {
        BAR_ S1,
        BAR_ I1
   } choiceId;
```

```
   union BarChoiceUnion
   {
        MyString s1; /* MyString */
       AsnInt i1; /* INTEGER */
   } a;
} Bar;
typedef MyString Bell; /* MyString */
```

The compiler directives used on the *MyString* type have some interesting effects. Notice that *MyString* is not referenced by pointer in the CHOICE, SET, or type definition, *Bell*. The generated code for encoding field *s1* of *Foo* type will use the code "*MYSTRING_NON_NULL (&fooVal->s1)*" to check for the presence of the OPTIONAL *s1* field. The code associated with MYSTRING_ NON_ NULL should return TRUE if the *s1* field value is present and might look like:

```
#define MYSTRING_ NON_ NULL( s) (* s != NULL)
```

The argument to *optTestRoutine* routine will be a pointer to the field type' s defining type. Note that in the above example, *MyString* is a *char **, therefore the *MYSTRING_ NON_ NULL* macro's argument will be a *char **.

Setting the *genPrintRoutine* etc. attributes to false makes eSNACC not define or generate any encode, decode, print, or free routines for the *MyString* type. You must provide these yourself; the best approach is to take the normal *PrintableString* routines and modify them to handle your special string type.

The names of the encode, decode, print and free routines used for the *MyString* type will be based on the ones given with the *printRoutineName* etc. attributes. eSNACC will prepend a "B" (for BER) and append a "Content" to the encode and decode routines names, so you must provide the *BEncMyStringContent* and *BDecMyStringContent* routines. You may also need the *BEncMyString* and *BDecMyString* routines if *MyString* is a PDU type or used in an ANY or ANY DEFINED type.

The *PrintableString* type has its C type name changed to *char **by the last compiler directive. Thus *MyString* is defined as a *char **. This directive applies to the *PrintableString* type reference. Note that these directives do not affect the tags or the encoded representation of the *MyString* type.

The location of the *–snacc* comment(s) is important. *–snacc* comment(s) between the *::=* sign and the following type are associated with the type being defined. Any compiler directives after the type and before the next type or value definition are

associated with the type. Fields in SETs, SEQUENCEs and CHOICEs can be modified by putting the compiler directive after the comma that follows the field type that you wish to modify. In the case of the last element of one of these types, where there is no comma, just place it after the field and before the closing bracket of the parent type.

Attributes shadow the type attributes filled in during the target language type information generation pass of the compiler. The type definition attributes are:

cTypeName this is the type name that the generated type will have. Its value can be any string that is valid as a C type name.

isPdu whether this is a PDU type. A PDU type will have extra interfaces to the encode and decode routines generated. Its value can be “TRUE” or “FALSE”.

isPtrForTypeDef TRUE if other types defined solely by this type definition are defined as a pointer to this type. Its value can be “TRUE” or “FALSE”.

isPtrForTypeRef TRUE if type references to this type definition from a SET or SEQUENCE are by pointer. Its value can be “TRUE” or “FALSE”.

isPtrInChoice TRUE if type references to this type definition from a CHOICE are by pointer. Its value can be “TRUE” or “FALSE”.

isPtrForOpt TRUE if OPTIONAL type references to this type definition from a SET or SEQUENCE are by pointer. Its value can be “TRUE” or “FALSE”.

optTestRoutineName name of the routine to test whether an OPTIONAL element of this type in a SET or SEQUENCE is present. The routine should return TRUE if the element is present. The value of this field is usually just the name of a C macro that tests for NON- NULL. The argument to the routine will be a pointer to the type definition's type. The optTestRoutineName value can be any string value.

defaultFieldName if this type is used in a SET, SEQUENCE or CHOICE without a field name then this value is used with a digit appended to it. Its value can be any string that is a valid C field name in a struct or union.

printRoutineName name of this type definition's printing routine. Its value can be any string that is a C function or macro name.

encodeRoutineName name of this type definition's encoding routine. Its value can be any string that is a C function or macro name.

decodeRoutineName name of this type definition's decoding routine. Its value can be any string that is a C function or macro name.

freeRoutineName name of this type definition's freeing routine. Its value can be any string that is a C function or macro name.

isEncDec If this type is used in a SET or SEQUENCE then it is not encoded or decoded. Its value can be “TRUE” or “FALSE”. This is handy for adding your own types to a standard that are only for local use, and are not included in encoded values.

genTypeDef TRUE if you want a C type to be generated for this type definition. Its values can be “TRUE” or “FALSE”.

genPrintRoutine TRUE if you want a printing routine to be generated for this type definition. Its values can be “TRUE” or “FALSE”.

genEncodeRoutine TRUE if you want an encoding routine to be generated for this type definition. Its values can be “TRUE” or “FALSE”.

genDecodeRoutine TRUE if you want a decoding routine to be generated for this type definition. Its values can be “TRUE” or “FALSE”.

genFreeRoutine TRUE if you want a free routine to be generated for this type definition. Its values can be “TRUE” or “FALSE”.

The type reference attributes are slightly different from the type definition attributes due to the semantic differences between a type definition and a type reference. Type references will inherit some of their attributes from the referenced type definition. The following are the valid type reference attributes:

cTypeName this is the type name that the generated type will have. Its value can be any string that is valid as a C type name.

cFieldName if this is a field in a CHOICE, SET or SEQUENCE then this holds the C field name for this reference. Its value can be any string that is valid as a C field name.

isPtr TRUE if this is a pointer to the type named by cTypeName. This is usually determined from the referenced type definitions attributes. Its value can be “TRUE” or “FALSE”.

optTestRoutineName if this field is an OPTIONAL component then this is the name of the routine to test whether it is present. The routine should return TRUE if the element is present. The value of this is usually just the name of a C macro that tests for NULL. The argument to the routine will be a pointer to the type

definition's type. The optTestRoutineName value can be any string value.

printRoutineName name of this type reference's printing routine. This and the other routine name attributes are useful for special instances of the referenced type. It is easier to modify the referenced type definition if you want every instance of this type to use a certain print etc. routine. Its value can be any string that is a value C function or macro name.

encodeRoutineName name of this type reference's encoding routine. Its value can be any string that is a function or macro name.

decodeRoutineName name of this type reference's decoding routine. Its value can be any string that is a C function or macro name.

freeRoutineName name of this type reference's freeing routine. Its value can be any string that is a C function or macro name.

isEncDec If this type is used in a SET or SEQUENCE then the field is not encoded or decoded. Its value can be “TRUE” or “FALSE”. This is handy for adding your own types to a standard that are only for local use, and are not included in encoded values.

choiceIdSymbol if this is a component of a CHOICE, this string attribute will be the defined/enum symbol whose value in the choiceId field indicates the presence of this field.

choiceIdValue if this is a component of a CHOICE, this integer attribute will be the value associated with the symbol in choiceIdSymbol.

2.10 Compiling the Generated C Code

The generated C code (and libraries) can be compiled by both ANSI and K& R C compilers. C function prototypes use the *PROTO* macro and C function declarations use the *PARAMS* macro. These macros are defined in `.../snacc.h` and their definitions depend on whether the *__USE_ANSI_C_* flag has been defined in `.../config.h`.

When compiling the generated C code you will need:

1. The include directory where the files from `.../c-lib/inc/` have been installed in your include path so the C sources can include the library header files. The header files should be included with statements like *#include <snacc/c/asn-incl.h>* and your C compiler should be supplied with

`-I/usr/local/include` in case eSNACC got installed under `/usr/local/`.

2. to link with the correct C ASN.1 runtime library, depending on the buffer type you choose. In case eSNACC got installed under `/usr/local/`, your linker may need to be supplied with `-L/usr/local/lib` and one of `-lasn1cebuf`, `-lasn1cmbuf` or `-lasn1csbuf` as arguments.

3. to link with the math library (`-lm`), since the ASN.1 REAL type's encode and decode routine use some math routines.

See the example in `.../c-examples/simple/` for a complete example. The makefile and main routines are probably the most important. There are several other examples in the `.../c-examples/`directory.

3 C ASN.1 Library

3.1 Overview

Each library type has a file in the `.../c-lib/src/` and `.../c-lib/inc/` directories. Each source file contains the encode, decode, free and print routines for the given type. This chapter contains a description of each library type and its routines. This library is also referred to as the runtime library.

After installing eSNACC, you should test the library types to make sure that they are encoding and decoding properly. Use the `.../c-examples/test-lib/` example to check them.

In addition to other errors, most decoding routines will report an error if they attempt to read past the end of the data. Be aware that some buffer types do not support this type of checking. This is explained more in the buffer management section.

3.2 Tags

eSNACC's tag representation was motivated by several things.

1. the tags must be easy to compare for equality in *if* and *switch* statements to make tag-based decisions cheap.
2. a tag must be cheap to decode.
3. a tag must be cheap to encode.

The first requirement meant that tags had to be integer types (for the *switch* statement). The representation of the tag within the integer was set by the second requirement.

The best way to decode cheaply is minimize the transformation between the encoded and decoded (internal) format. So the four (can be set-up for two) bytes of the long integer are used to hold the encoded tag, starting with the first octet of the tag in the most significant byte of the integer and the rest (if any) following. Any unused (always trailing) bytes in the integer are zero. This limits the representable tag code to less than 2^{21} but for reasonable ASN.1 specifications this should not be a problem.

To meet the third requirement the decoded tag representation was bypassed entirely by using macros (*BEncTag1()* etc.) that write the encoded tag octet(s) to the buffer. The writing of an encoded tag octet involves bit shifting, bitwise ands and bitwise ors with constant values; most optimizing C compilers can compute these at

compile time. This simplifies encoding a tag to writing some constant byte value(s) to the buffer.

The following excerpt from `.../c-lib/inc/asn-tag.h` shows some of the tag routines.

```
typedef unsigned long int AsnTag;
#define MAKE_ TAG_ ID( class, form, code) ...
#define TAG_ IS_ CONS( tag) ...
#define BEncTag1( b, class, form, code) ...
#define BEncTag2( b, class, form, code) ...
#define BEncTag3( b, class, form, code) ...
#define BEncTag4( b, class, form, code) ...
#define BEncTag5( b, class, form, code) ...
AsnTag BDecTag (BUF_ TYPE b, AsnLen *bytesDecoded, ENV_ TYPE env);
```

The generated decode routines use the *BDecTag* to decode a tag from the buffer. The returned tag value is either used in an *if* expression or as the argument to *switch* statements. The *MAKE_TAG_ID* macro is used to make a tag for comparison to the one returned by *BDecTag*. The *MAKE_TAG_ID* is used in *switch* statement case labels and in *if* statements.

Most of the time tags are only compared for equality, however, the OCTET STRING and BIT STRING decoders check the constructed bit in the tag using the *TAG_IS_CONS* macro.

The *BEncTag* macros are quite fragile because they return the encoded length of the tag; they cannot be treated as a single statement. This requires careful use of braces when using them in your own code in places such as the sole statement in an *if* statement. This ugliness is caused by the difficulty in returning values from multi-line macros (macros are used for performance here since encoding tags can be a significant part of BER encoding).

The *BDecTag* routine will report an error via *longjmp* if the encoded tag is longer than can be held in the *AsnTag* type or if it read past the end of the data when decoding the tag.

3.3 Lengths

Decoded lengths are represented by unsigned long integers, with the maximum value indicating indefinite length.

eSNACC users can choose between using only indefinite or only definite lengths when encoding constructed values' lengths when compiling the generated code. Of course, the generated decoders can handle both forms. Define the *USE_INDEF_LEN* symbol when compiling the generated code if you want to use indefinite lengths when encoding constructed values. Primitive values are always encoded with definite lengths as required by the standard; this is necessary to avoid confusion between a value's content and the End-Of-Contents marker.

There is no loss of performance when using definite lengths with eSNACC encoders. This is due the “backwards” encoding as described in Section 4.4. The schemes used by other compilers' encoders to handle definite lengths may hurt performance. Most of the routines in the following code are obvious except for *BEncDefLenTo127()*. This is used instead of *BEncDefLen* in the generated code when the compiler knows the value being encoded will not be over 127 octets long. Values such as BOOLEANs, INTEGERs, and REALs are assumed to be shorter than 127 octets (constraints on the decoded representation of INTEGERs and REALs make this valid).

```
typedef unsigned long int AsnLen;

/* max unsigned value - used for internal rep of indef len */

#define INDEFINITE_LEN ~0L

#ifdef USE_INDEF_LEN

#define BEncEocIfNec( b)      BEncEoc (b)

#define BEncConsLen( b, len)  2 + BEncIndefLen (b)

#else

#define BEncEocIfNec( b)

#define BEncConsLen( b, len)  BEncDefLen (b, len)

#endif

#define BEncIndefLen( b) ...

#define BEncDefLenTo127( b, len) ...

AsnLen BEncDefLen (BUF_TYPE b, AsnLen len);

AsnLen BDecLen (BUF_TYPE b, AsnLen *bytesDecoded, ENV_TYPE env);

#define BEncEoc( b) ...

#define BDEC_2ND_EOC_OCTET( b, bytesDecoded, env) ...

void BDecEoc (BUF_TYPE b, AsnLen *bytesDecoded, ENV_TYPE env);
```

The BDecLen routine will report an error via longjmp if it attempts to read past the end of the data or the decoded length is too large

to be held in the AsnLen representation. BDecEoc will report an error if it attempts to read past the end of the data or one of the EOC (End- Of- Contents) octets is non- zero.

3.4 BOOLEAN

The BOOLEAN type is represented by an *unsigned char*. It has the following routines for manipulating it.

```
typedef unsigned char AsnBool;

AsnLen BEncAsnBool (BUF_TYPE b, AsnBool *data);

void BDecAsnBool (BUF_TYPE b, AsnBool *result, AsnLen *bytesDecoded,
ENV_TYPE env);

AsnLen BEncAsnBoolContent (BUF_TYPE b, AsnBool *data);

void BDecAsnBoolContent (BUF_TYPE b, AsnTag tag, AsnLen len, AsnBool
*result, AsnLen                                    *bytesDecoded,
ENV_TYPE env);

#define FreeAsnBool( v)

void PrintAsnBool (FILE *f, AsnBool *b, unsigned short int indent);
```

As discussed in Sections 4.4 and 4.5, *BEncAsnBool* and *BDecAsnBool* encode/decode the UNIVERSAL tag, length and content of the given BOOLEAN value. The *BEncAsnBoolContent* and *BDecAsnBoolContent* routine only encode/decode the content of the given BOOLEAN value.

The *FreeAsnBool* routine does nothing since the BOOLEAN type does not contain pointers to data; the free routine generator does not have to check which types need freeing and simply calls the type's free routine. It also allows the user to modify the types and their free routines without changing the free routine generator. However, the ANY and ANY DEFINED BY type hash table initialization routine generator does need to know which types have empty free routines because the hash entries contain pointers to the free functions (NULL is used for the empty free functions like *FreeAsnBool*). The INTEGER, NULL, REAL and ENUMERATED types have empty free routines for the same reason.

BDecAsnBool will report an error if the tag is not UNIVERSAL-PRIM-1. *BDecAsnBoolContent* will report an error if it decodes past the end of the data or the length of the encoded value (given by the *len* parameter) is not exactly one octet.

3.5 INTEGER

The INTEGER type is represented by a 32 bit integer type, *AsnInt*. The C integer type chosen depends on the machine and compiler and may be *int*, *long* or *short*, whatever is 32 bits in size. If you are using INTEGER types that are only positive (via subtyping or protocol definition) you may want to use the *UAsnInt* and associated routines that use the unsigned int for a larger positive value range.

```
typedef int AsnInt;

typedef unsigned int UAsnInt;

AsnLen BEncAsnInt (BUF_TYPE b, AsnInt *data);

void BDecAsnInt (BUF_TYPE b, AsnInt *result, AsnLen *bytesDecoded,
ENV_TYPE env);

AsnLen BEncAsnIntContent (BUF_TYPE b, AsnInt *data);

void BDecAsnIntContent (BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
AsnInt *result, AsnLen *bytesDecoded, ENV_TYPE env);

#define FreeAsnInt( v)

void PrintAsnInt (FILE *f, AsnInt *v, unsigned short int indent);

AsnLen BEncUAsnInt (BUF_TYPE b, UAsnInt *data);

void BDecUAsnInt (BUF_TYPE b, UAsnInt *result, AsnLen *bytesDecoded,
ENV_TYPE env);

AsnLen BEncUAsnIntContent (BUF_TYPE b, UAsnInt *data);

void BDecUAsnIntContent (BUF_TYPE b, AsnTag tagId, AsnLen len,
UAsnInt *result, AsnLen *bytesDecoded, ENV_TYPE env);

#define FreeUAsnInt( v)

void PrintUAsnInt (FILE *f, UAsnInt *v, unsigned short int indent);
```

BDecAsnInt will report an error if the tag is not UNIVERSAL-PRIM-2 *BDecAsnIntContent* will report an error if it decodes past the end of the data or the integer value is too large for an *AsnInt.*

3.6 NULL

The NULL type is represented by the *AsnNull* type. Its content is always empty and hence its encoded length always is zero.

```
typedef char AsnNull;

AsnLen BEncAsnNull (BUF_TYPE b, AsnNull *data);

void BDecAsnNull (BUF_TYPE b, AsnNull *result, AsnLen *bytesDecoded,
ENV_TYPE env);

/* 'return' length of encoded NULL value, 0 */

#define BEncAsnNullContent( b, data)     0
```

```
void BDecAsnNullContent (BUF_TYPE b, AsnTag tag, AsnLen len, AsnNull
*result, AsnLen *bytesDecoded, ENV_TYPE env);

#define FreeAsnNull( v)

void PrintAsnNull (FILE *f, AsnNull * b, unsigned short int indent);
```

3.7 REAL

The REAL type is represented by *AsnReal,* a double. This type's representation can depend on the compiler or system you are using so several different encoding routines are provided. Even so, you may need to modify the code.

If you are using the REAL type in your ASN.1 modules, you should call the *InitAsnInfinity()* routine to setup the *PLUS_INFINITY* and *MINUS_INFINITY* values. There are three encode routines included and they can be selected by defining one of *IEEE_REAL_FMT*, *IEEE_REAL_LIB* or nothing. Defining *IEEE_REAL_FMT* uses the encode routine that assumes the double representation is the standard IEEE double [3]. Defining *IEEE_REAL_LIB* uses the encode routine that assumes the IEEE functions library (isinf, scalbn, signbit etc.) is available. If neither are defined, the default encode routine uses *frexp*.

There is only one content decoding routine and it builds the value through multiplication and the *pow* routine (requires the math library). The content decoding routine only supports the binary encoding of a REAL, not the decimal encoding.

```
typedef double AsnReal;

extern AsnReal PLUS_INFINITY;

extern AsnReal MINUS_INFINITY;

void InitAsnInfinity();

AsnLen BEncAsnReal (BUF_TYPE b, AsnReal *data);

void BDecAsnReal (BUF_TYPE b, AsnReal *result, AsnLen *bytesDecoded,
ENV_TYPE env);

AsnLen BEncAsnRealContent (BUF_TYPE b, AsnReal *data);

void BDecAsnRealContent (BUF_TYPE b, AsnTag tag, AsnLen len, AsnReal
*result, AsnLen *bytesDecoded, ENV_TYPE env);

/* do nothing */

#define FreeAsnReal( v)

void PrintAsnReal (FILE *f, AsnReal *b, unsigned short int indent);
```

BDecAsnReal will report an error if the value's tag is not UNIVERSAL-PRIM-9. *BDecAsnReaContentl* will report an error if

the base is not supported or the decimal type REAL encoding is received.

3.8 BIT STRING

The BIT STRING type is represented by the *AsnBits* structure. It contains a pointer to the bits and integer that holds the length in bits of the BIT STRING.

In addition to the standard encode, decode, print and free routines, there are some other utility routines. *AsnBitsEquiv* returns TRUE if the given BIT STRINGs are identical. The *SetAsnBit*, *ClrAsnBit* and *GetAsnBit* are routines for writing and reading a BIT STRING value.

You may notice that the AsnBits type does not have any means of handling linked pieces of BIT STRINGs. Some ASN.1 tools use lists of structures like *AsnBits*to represent BIT STRINGs. This is done because, as you should be aware, BIT STRINGs can be encoded in a nested, constructed fashion. The eSNACC BIT STRING decoder attempts to save you the hassle of dealing with fragments of BIT STRINGs by concatenating them in the decoding step. Every BIT STRING value returned by the decoder will have contiguous bits.

Some people contend that fragmented BIT STRINGs are necessary to support systems that lack enough memory to hold the entire value. eSNACC encodes value “backwards” so the entire value must be encoded before it can be sent, thus you must have enough memory to hold the whole encoded value. If the fragmented representation is useful to your protocol implementation for other reasons, it should be fairly simple to modify the BIT STRING routines. Remember, no significance should be placed on where constructed BIT STRING values are fragmented.

eSNACC uses a table to hold pointers to the BIT STRING fragments in the buffer while it is decoding them. Once the whole BIT STRING value has been decoded, a block of memory that is large enough to hold the entire BIT STRING is allocated and the fragments are copied into it. The table initially can hold pointers to 128 fragments. If more table entries are needed the stack will grow via *realloc* (with associated performance loss) and will not shrink after growing. If you wish to modify this behavior, change the `.../c-lib/inc/str-stk.h` file.

The *FreeAsnBits* routine will free memory referenced by the *bits* pointer.

```
typedef struct AsnBits
```

```
{
        int bitLen;
        char *bits;
} AsnBits;
extern char numToHexCharTblG[];
#define TO_HEX( fourBits)           (numToHexCharTblG[( fourBits) &
0x0f])
#define ASNBITS_PRESENT( abits)     (( abits)-> bits != NULL)
AsnLen BEncAsnBits (BUF_TYPE b, AsnBits *data);
void BDecAsnBits (BUF_TYPE b, AsnBits *result, AsnLen *bytesDecoded,
ENV_TYPE env);
AsnLen BEncAsnBitsContent (BUF_TYPE b, AsnBits *bits);
void BDecAsnBitsContent (BUF_TYPE b, AsnLen len, AsnTag tagId,
AsnBits *result, AsnLen *bytesDecoded, ENV_TYPE env);
void FreeAsnBits (AsnBits *v);
void PrintAsnBits (FILE *f, AsnBits *b, unsigned short int indent);
int AsnBitsEquiv (AsnBits *b1, AsnBits *b2);
void SetAsnBit (AsnBits *b1, unsigned long int bit);
void ClrAsnBit (AsnBits *b1, unsigned long int bit);
int GetAsnBit (AsnBits *b1, unsigned long int bit);
```

BDecAsnBits will report an error if the tag is not UNIVERSAL-CONS-3 or UNIVERSAL-PRIM-3. When decoding constructed BIT STRING BER values, an error will be reported if a component other than the last one has non-zero unused bits in its last octet or an internal component does not have the UNIVERSAL-3 tag. If the decoder attempts to read past the end of the data an error will be reported.

3.9 OCTET STRING

The OCTET STRING type is represented by the *AsnOcts* structure. It contains a pointer to the octets and an integer that holds the length in octets of the OCTET STRING.

As with BIT STRINGs, OCTET STRINGs can have constructed values. These are handled in the same way as the constructed BIT STRING values. The decoded representation of an OCTET STRING is always contiguous.

The *FreeAsnOcts* routine will free the memory referenced by the *octs* pointer. The *AsnOctsEquiv* routine will return TRUE if the given OCTET STRINGs are identical.

```
typedef struct AsnOcts
{
        unsigned long int octetLen;
        char *octs;
} AsnOcts;
#define ASNOCTS_PRESENT( aocts)      (( aocts)-> octs != NULL)
AsnLen BEncAsnOcts (BUF_TYPE b, AsnOcts *data);
void BDecAsnOcts (BUF_TYPE b, AsnOcts *result, AsnLen *bytesDecoded,
ENV_TYPE env);
AsnLen BEncAsnOctsContent (BUF_TYPE b, AsnOcts *octs);
void BDecAsnOctsContent (BUF_TYPE b, AsnLen len, AsnTag tagId,
AsnOcts *result, AsnLen *bytesDecoded, ENV_TYPE env);
void FreeAsnOcts (AsnOcts *o);
void PrintAsnOcts (FILE *f, AsnOcts *o, unsigned short int indent);
int AsnOctsEquiv (AsnOcts *o1, AsnOcts *o2);
```

BDecAsnOcts will report an error if the tag is not UNIVERSAL-CONS-4 or UNIVERSAL-PRIM-4. When decoding constructed OCTET STRING BER values, an error will be reported if an internal component does not have the UNIVERSAL-4 tag. If the decoder attempts to read past the end of the data an error will be reported.

3.10 Built-in Strings PrintableString, BMPString, TeletexString, NumericString, IA5String, UniversalString, UTF8String, VisibleString

The eSNACC run-time library directly supports the various ASN.1 string definitions. This includes validating the actual data restrictions for the various ASN.1 string types (i.e. encode operations will fail if invalid characters are being encoded). All strings support some basic load/unload operations. Some of the strings will handle wide characters (e.g. BMPString, UTF8String). For example, in “asn-UTF8String.h”:

```
int CvtUTF8String2wchar(UTF8String *inOcts, wchar_t **outStr);
int CvtUTF8towchar(char *utf8Str, wchar_t **outStr);
int CvtWchar2UTF8(wchar_t *inStr, char **utf8Str);
```

3.11 OBJECT IDENTIFIER

In eSNACC, OBJECT IDENTIFIERs are kept in their encoded form to improve performance. The *AsnOid* type is defined as *AsnOcts*, as it holds the octets of the encoded OBJECT IDENTIFIER. It seems that the most common operation with OBJECT IDENTIFIERs is to compare for equality, for which the encoded representation (which is canonical) works well.

There is a linked OBJECT IDENTIFIER representation called *OID* and routines to convert it to and from the *AsnOid* format, but it should not be used if performance is an issue.

Since the OBJECT IDENTIFIERs are represented *AsnOcts*, the *AsnOcts* content encoding routine can be used for the *AsnOid* content encoding routine. The other *AsnOcts* encoding and decoding routines cannot be used because the OBJECT IDENTIFIER has a different tag and cannot be encoded in a constructed fashion.

An OBJECT IDENTIFIER must have a minimum of two arc numbers but the decoding routines do not check this.

RELATIVE OIDs have also been added. They are stored in there encoded form as well, but are not required to have at least two arc numbers. RELATIVE OIDs must be associated with a root OBJECT IDENTIFIER.

```
typedef AsnOcts AsnOid;

#define ASNOID_PRESENT( aoid)       ASNOCTS_PRESENT (aoid)

AsnLen BEncAsnOid (BUF_TYPE b, AsnOid *data);

void BDecAsnOid (BUF_TYPE b, AsnOid *result, AsnLen *bytesDecoded,
ENV_TYPE env);

#define BEncAsnOidContent( b, oid)  BEncAsnOctsContent( b, oid)

void BDecAsnOidContent (BUF_TYPE b, AsnTag tag, AsnLen len, AsnOid
*result, AsnLen *bytesDecoded, ENV_TYPE env);

#define FreeAsnOid FreeAsnOcts

void PrintAsnOid (FILE *f, AsnOid *b, unsigned short int indent);

#define AsnOidsEquiv( o1, o2)       AsnOctsEquiv (o1, o2)
```

3.12 SET OF and SEQUENCE OF

The SET OF and SEQUENCE OF type are represented by the *AsnList* structure. An *AsnList* consists of a head object that has pointers to the first, current and last nodes and the current number of nodes in the list. Each list node has a pointer to its next and previous list member and the node's data. The first list node's

previous pointer is always NULL and the last list node's next pointer is always NULL.

Each SET OF or SEQUENCE OF type is defined as an *AsnList*, so the element type information (kept via a *void **) is not kept, therefore, the *AsnList* type is not type safe.

The *AsnList* is a doubly linked list to simplify “backwards” encoding. The reverse link allows the list to be traversed in reverse so the components can be encoded from last to first.

Initially, the lists were designed to allow the list element itself to be contained in the list node (hence the *elmtSize* parameter to the AsnListNew() routine). The design eventually changed such that every list element was reference by pointer from the list node.

A small problem with the *AsnListNew* routine is the memory allocation. Since it is used by the decoding routines to allocate new lists, it uses whatever memory management you have setup with the *Asn1Alloc* macro (see Section 5.14). This may not be desirable when building values to be transmitted. You may need to provide another AsnListNew routine that uses a different allocation scheme to solve this.

```
typedef struct AsnListNode
{
        struct AsnListNode *prev;
        struct AsnListNode *next;
        void *data; /* this must be the last field of this structure
*/
} AsnListNode;
typedef struct AsnList
{
        AsnListNode *first;
        AsnListNode *last;
        AsnListNode *curr;
        int count; /* number of elements in list */
        int dataSize; /* space required in each node for the data */
} AsnList;
#define FOR_EACH_LIST_ELMT( elmt, list) ...
#define FOR_EACH_LIST_ELMT_RVS( elmt, list) ...
#define FOR_REST_LIST_ELMT( elmt, al) ...
#define CURR_LIST_ELMT( al)   (al)-> curr-> data
#define NEXT_LIST_ELMT( al)   (al)-> curr-> next-> data
```

```
#define PREV_LIST_ELMT( al)   (al)-> curr-> prev-> data
#define LAST_LIST_ELMT( al)   (al)-> last-> data
#define FIRST_LIST_ELMT( al)  (al)-> first-> data
#define LIST_EMPTY( al)  (( al)-> count == 0)
#define CURR_LIST_NODE( al)   (( al)-> curr)
#define FIRST_LIST_NODE( al)  (( al)-> first)
#define LAST_LIST_NODE( al)   (( al)-> last)
#define PREV_LIST_NODE( al)   (( al)-> curr-> prev)
#define NEXT_LIST_NODE( al)   (( al)-> curr-> next)
#define SET_CURR_LIST_NODE( al, listNode)       (( al)-> curr =
(listNode))
void AsnListRemove (AsnList *l);
void *AsnListAdd (AsnList *l);
void *AsnListInsert (AsnList *list);
void AsnListInit (AsnList *list, int dataSize);
AsnList *AsnListNew (int elmtSize);
void *AsnListPrev (AsnList *);
void *AsnListNext (AsnList *);
void *AsnListLast (AsnList *);
void *AsnListFirst (AsnList *);
void *AsnListPrepend (AsnList *);
void *AsnListAppend (AsnList *);
void *AsnListCurr (AsnList *);
int AsnListCount (AsnList *);
AsnList *AsnListConcat (AsnList *, AsnList *);
```

There are a number of macros for dealing with the list type, the most important being the list traversal macros. The *FOR_EACH_LIST_ELMT* macro acts like a “for” statment that traverses forward through the list. The first parameter should be a pointer to the list element type that will be used to hold the current list element for each iteration of the “for” loop. The second parameter is the list of elements that you wish to traverse.

The *FOR_EACH_LIST_ELMT_RVS* macro is identical to the *FOR_EACH_LIST_ELMT* macro except that is moves from the back of the list to the front. The *FOR_REST_LIST_ELMT* macro is similar to the other two but it does not reset the *curr* pointer in the *AsnList* type. This has the effect of iterating from the current element to the end of the list. Look in the generated code for a

better indication of how to use these macros. The other macros are straight forward.

3.13 ANY and ANY DEFINED BY

The ANY and ANY DEFINED BY type are classically the most irritating ASN.1 types for compiler writers. They rely on mechanisms outside of ASN.1 to specify what types they contain. The 1992 ASN.1 standard has rectified this by adding much stronger typing semantics and eliminating macros.

The ANY DEFINED BY type can be handled automatically by *eSNACC* if the SNMP OBJECT-TYPE [10] macro is used to specify the identifier value to type mappings. The identifier can be an INTEGER or OBJECT IDENTIFIER. Handling ANY types properly will require modifications to the generated code since there is no identifier associated with the type.

The general approach used by *eSNACC* to handle ANY DEFINED BY types is to lookup the identifier value in a hash table for the identified type. The hash table entry contains information about the type such as the routines to use for encoding and decoding.

Two hash tables are used, one for INTEGER to type mappings and the other for OBJECT IDENTIFIER to type mappings. *eSNACC* generates an `InitAny` routine for each module that uses the OBJECT-TYPE macro. This routine adds entries to the hash table(s). The `InitAny` routine(s) is called once before any encoding or decoding is done.

The hash tables are constructed such that an INTEGER or OBJECT IDENTIFIER value will hash to an entry that contains:

- the `anyId`
- the INTEGER or OBJECT IDENTIFIER that maps to it
- the size in bytes of the identified data type
- a pointer to the type's PDU encode routine
- a pointer to the type's PDU decode routine
- a pointer to the type's print routine
- a pointer to the type's free routine

The referenced encode and decode routines are PDU oriented in that they encode the type's tag(s) and length(s) as well as the type's content.

eSNACC builds an `enum` called `AnyId` that enumerates each mapping defined by the OBJECT-TYPE macros. The name of the value associated with each macro is used as part of the enumerated identifier. The `anyId` in the hash table holds the identified type's `AnyId enum` value. The `anyId` is handy for making decisions based on the received identifier, without comparing OBJECT IDENTIFIERs. If the identifiers are INTEGERs then the `anyId` is less useful.

With ANY DEFINED BY types, it is important to have the identifier decoded before the ANY DEFINED BY type is decoded. Hence, an ANY DEFINED BY type should not be declared before its identifier in a SET since SETs are un-ordered. An ANY DEFINED BY type should not be declared after its identifier in a SEQUENCE. *eSNACC* will print a warning if either of these situations occur.

The hash tables may be useful to plain ANY types which do not have an identifier field like the ANY DEFINED BY types; the OBJECT-TYPE macro can be used to define the mappings and the `SetAnyTypeByInt` or `SetAnyTypeByOid` routine can be called with the appropriate identifier value before encoding or decoding an ANY value.

OPTIONAL ANYs and ANY DEFINED BY types that have not been tagged are a special problem for *eSNACC* . Unless they are the last element of a SET or SEQUENCE, the generated code will need to be modified. *eSNACC* will print a warning message when it encounters one of these cases.

To illustrate how ANY DEFINED BY values are handled, we present typical encoding and decoding scenarios. Each ANY or ANY DEFINED BY type is represented in C by the `AsnAny` type which contains only a `void *` named `value` to hold a pointer to the value and a `AnyInfo *` named `ai` which points to a hash table entry.

When encoding, before the ANY DEFINED BY value is encoded, `SetAnyTypeByOid` or `SetAnyTypeByInt` (depending on the type of the identifier) is called with the current identifier value to set the `AsnAny` value' s `ai` pointer to the proper hash table entry. Then to encode the ANY DEFINED BY value, the encode routine pointed to from the hash table entry is called with the `value void *` from the `AsnAny` value. The `value void *` in the `AsnAny` should point to a value of the correct type for the given identifier, if the user set it up correctly.

Note that setting the `void *` value is not type safe; one must make sure that the value' s type is the same as indicated by the identifier.

For decoding, the identifier must be decoded prior to the ANY DEFINED BY value otherwise the identifier will contain an uninitialized value. Before the ANY or ANY DEFINED BY value is decoded, `SetAnyTypeByOid` or `SetAnyTypeByInt` (de- pending on the type of the identifier) is called to set the `AsnAny` value' s `ai` pointer to the proper hash table entry. Then a block of memory of the size indicated in the hash table entry is allocated, and its pointer stored in the `AsnAny` value' s `void*` entry. Then the decode routine pointed to from the hash table entry is called with the newly allocated block as its value pointer parameter. The decode routine fills in the value assuming it is of the correct type. Simple!

There is a problem with *eSNACC* 's method for handling ANY DEFINED BY types for specifications that have two or more ANY DEFINED BY types that share some identifier values. Since only two hash tables are used and they are referenced using the identifier value as a key, duplicate identifiers will cause unresolvable hash collisions.

Here is some of the *AsnAny* related code from the header file. It should help you understand the way things are done a bit better. Look in the `hash.c` and `hash.h` files as well.

```
/*
 * 1 hash table for integer keys
 * 1 hash table for oid keys
 */
extern Table *anyOidHashTblG;
extern Table *anyIntHashTblG;
typedef (* EncodeFcn) (BUF_TYPE b, void *value);
typedef void (* DecodeFcn) (BUF_TYPE b, void *value, AsnLen
*bytesDecoded, ENV_TYPE env);
typedef void (* FreeFcn) (void *v);
typedef void (* PrintFcn) (FILE *f, void *v);
/*
 * this is put into the hash table with the
 * int or oid as the key
 */
typedef struct AnyInfo
{
        int anyId; /* will be a value from the AnyId enum */
```

```
        AsnOid oid; /* will be zero len/ null if intId is valid */
        AsnInt intId;
        unsigned int size; /* size of the C data type (ie as ret'd by
sizeof) */
        EncodeFcn Encode;
        DecodeFcn Decode;
        FreeFcn Free; PrintFcn Print;
} AnyInfo;
typedef struct AsnAny
{
        AnyInfo *ai; /* point to entry in hash tbl that has routine
ptrs */
        void *value; /* points to the value */
} AsnAny;
/*
  * Returns anyId value for the given ANY type.
  * Use this to determine to the type of an ANY after decoding
  * it. Returns -1 if the ANY info is not available
  */
#define GetAsnAnyId( a)  ((( a)-> ai)? (a)-> ai-> anyId: -1)
/*
  * used before encoding or decoding a type so the proper
  * encode or decode routine is used.
  */
void SetAnyTypeByInt (AsnAny *v, AsnInt id);
void SetAnyTypeByOid (AsnAny *v, AsnOid *id);
/*
  * used to initialize the hash table(s)
  */
void InstallAnyByInt (int anyId, AsnInt intId, unsigned int size,
EncodeFcn encode, DecodeFcn decode, FreeFcn free, PrintFcn print);
void InstallAnyByOid (int anyId, AsnOid *oid, unsigned int size,
EncodeFcn encode, DecodeFcn decode, FreeFcn free, PrintFcn print);
/*
  * Standard enc, dec, free, & print routines.
  * for the AsnAny type.
  * These call the routines referenced from the
  * given value's hash table entry.
```

```
  */
void FreeAsnAny (AsnAny *v);
AsnLen BEncAsnAny (BUF_TYPE b, AsnAny *v);
void BerDecAsnAny (BUF_TYPE b, AsnAny *result, AsnLen *bytesDecoded,
ENV_TYPE env);
void PrintAsnAny (FILE *f, AsnAny *v, unsigned short indent);
/* AnyDefinedBy is the same as AsnAny */
typedef AsnAny AsnAnyDefinedBy;
#define FreeAsnAnyDefinedBy FreeAsnAny
#define BEncAsnAnyDefinedBy BEncAsnAny
#define BDecAsnAnyDefinedBy BDecAsnAny
#define PrintAsnAnyDefinedBy PrintAsnAny
```

3.13.1 ANY Automatic Buffer Handling

To handle general “ANY” elements, the eSNACC run-time library and compiler have been updated to process these elements as general buffers. The example program demonstrates how to load/unload the ANY data. It is assumed that the user has appropriately encoded the data being loaded into the ANY, no data checking is performed. On decode, the appropriate length (from the tag/length) is determined and unloaded into the buffer. The decode operation will fail if the buffer is not properly ASN.1 encoded (with an associated tag/length). It is no longer necessary to hand-edit the eSNACC generated files for a proper “C” build. The following code snippet demonstrates this feature; this logic is presented in the “C” examples directory.

```
struct ExpBuf *pBUF, *pBUF2=NULL;
struct GenBuf *gBUF, *gBUF2=NULL;
PrintableString *pB;   // FIRST, load PrintableString for ANY.
pBUF = ExpBufAllocBufAndData();
ExpBuftoGenBuf(pBUF, &gBUF);
pB = (PrintableString *) calloc(1, sizeof(PrintableString));
pB->octs = strdup(test);
pB->octetLen = strlen(test);
ExpBufResetInWriteRvsMode(gBUF->spare);
result = BEncPrintableString(gBUF, pB);
FreePrintableString(pB);
free(pB);
ExpBufResetInReadMode(gBUF->bufInfo);
```

```
   if (result)
   {
       // NEXT, load encoded PrintableString (in gBUF) into ANY of A4
       memset(&A4,'\0', sizeof(A4)); // A4.aBlob is an AsnAny
       iAnyLen = 0;
       SetAnyTypeUnknown((&A4.aBlob));  // SETUP for this buffer of ANY...
       BDecAsnAny(gBUF, &A4.aBlob, &iAnyLen, env);
       if (iAnyLen)
          …<<<< SUCCESSFUL ANY load >>>>
…
       SetAnyTypeUnknown((&A5.aBlob));  // SETUP for this buffer of ANY...
       ExpBufResetInWriteRvsMode(gBUF->spare);
       iAnyLen = BEncAsnAny(gBUF, &A5.aBlob);
       if (iAnyLen)
       {
           pPS = (PrintableString *) calloc(1, sizeof(PrintableString));
           iAnyLen = 0;   //THE application would have to know the data-type to decode.
           ExpBufResetInReadMode(gBUF->bufInfo);
           BDecPrintableString(gBUF, pPS, &iAnyLen, env); // WILL FAIL if ANY is not PrintableString.
           if (iAnyLen)  // SUCCEEDED in deocding ANY, previously encoded
          …<<<< SUCCESSFUL ANY unload >>>>
…
```

3.14 Buffer Management

Encoding and decoding performance is heavily affected by the cost of writing to and reading from buffers, thus, efficient buffer management is necessary. Flexibility is also important to allow integration of the generated encoders and decoders into existing environments. To provide both of these features, the calls to the buffer routines are actually macros that can be configured as you want (see `.../c-lib/inc/asn-config.h`). Virtually all buffer calls will be made from the encode/decode library routines. So macros used in the generated code will make buffer calls.

If your environment uses a single, simple buffer type, the buffer routine macros can be defined as the macros for your simple buffer type. This results in the buffer type being bound at compile time, with no function call overhead from the encode or decode routines. This also means that the runtime library only works for that buffer type.

If multiple buffer formats must be supported at runtime, the buffer macros can be defined like the ISODE buffer calls, where a buffer type contains pointers to the buffer routines and data of the current buffer type. This approach will hurt performance since each buffer operation will be an indirect function call. I have implemented buffers like this for the table tools (performance is

already hosed so slower buffer routines are a drop in the bucket). See the type tables section for their description.

The backwards encoding technique requires special buffer primitives that write from the end of the buffer towards the front. This requirement will make it impossible to define buffer primitives that write directly to stream oriented objects such as TCP connections. In cases such as this, you must encode the entire PDU before sending it. (Or else extend the back-end of the compiler to produce “forwards” encoders as well).

Nine buffer primitives are required by the runtime library's encode and decode routines:

- unsigned char BufGetByte(BUF_TYPE b);
- unsigned char BufPeekByte(BUF_TYPE b);
- char *BufGetSeg(BUF_TYPE b, unsigned long int *lenPtr);
- char BufCopy(char *dst, BUF_TYPE b, unsigned long int *lenPtr);
- void BufSkip (BUF_TYPE b, unsigned long int len);
- void BufPutByteRv (BUF_TYPE b, unsigned char byte);
- void BufPutSegRv (BUF_TYPE b, char *data, unsigned long int len);
- int BufReadError (BUF_TYPE b);
- int BufWriteError (BUF_TYPE b);

These buffer operations are described in the next subsections. The *ExpBuf*, *SBuf* and *MinBuf* buffer formats that come with the eSNACC distribution and how to configure the buffer operations are discussed following that.

3.14.1 Buffer Reading Routine Semantics

The buffer reading routines are called by the decoder routines. The following is the list of necessary buffer reading routines and their semantics. Be sure to setup the buffer in reading mode before calling any of these routines. The means of putting a buffer in reading mode depends on the buffer type.

```
unsigned char BufGetByte (BUF_TYPE b);
```

Returns the next byte from the buffer and advances the current pointer such that a subsequent buffer read returns the following byte(s). This will set the read error flag if an attempt to read past the end of the data is made.

```
unsigned char BufPeekByte (BUF_TYPE b);
```

Returns the next byte from the buffer without advancing the current pointer.

```
char *BufGetSeg (BUF_TYPE b, unsigned long int *lenPtr);
```

Returns a pointer to the next bytes from the buffer and advances the current pointer. **lenPtr* should contain the number of bytes to read. If the buffer has a least **lenPtr* contiguous bytes remaining to be read before calling *BufGetSeg*, a pointer to them will be returned and **lenPtr* will be unchanged. If there are less than **lenPtr* contiguous bytes remaining in the buffer before the call to *BufGetSeg*, a pointer to them is returned and **lenPtr* is set to the actual number of bytes that are referenced by the returned pointer. The current pointer will be advanced by the value returned in **lenPtr* (this may advance to the next buffer segment if any). Note that the read error flag is not set if **lenPtr* is greater than the remaining number of unread bytes.

```
unsigned long int BufCopy (char *dst, BUF_TYPE b,
unsigned long int len)
```

Copies the next *len* bytes from the buffer into the *dst char **and advances the current pointer appropriately. Returns the number of bytes actually copied. The number of bytes copied will be less than requested only if the end of data is reached, in which case the read error flag is set.

```
void BufSkip (BUF_TYPE b, unsigned long int len);
```

Advances the buffer's current pointer by *len* bytes. This will set the read error flag if less than *len* unread bytes remain in the buffer before the call to *BufSkip*.

```
int BufReadError (BUF_TYPE b);
```

Returns non-zero if a read error occurred for the given buffer. Read errors occur if one of the buffer reading routines attempted to read past the end of the buffer's data.

3.14.2 Buffer Writing Routine Semantics

Encoding routines call the buffer writing routines. Here is a list of the buffer writing routine and their semantics. Before calling the writing routines, you should make sure the buffer is setup for writing in reverse mode. The means of doing this depends on the buffer type.

```
void BufPutByteRvs (BUF_TYPE b, unsigned char byte);
```

Writes the given byte to the beginning of the data in the given buffer. The newly written byte becomes part of the buffer's data

such that subsequent writes place bytes before the newly written byte. If a buffer write error occurs, subsequent writes do nothing.

```
void BufPutSegRvs (BUF_TYPE b, char *data, unsigned long int len);
```

Prepends the given bytes, *data,* of length *len* to the beginning of the data in the given buffer *b*. The *data* bytes are written such that the first byte in *data* becomes the first byte of the buffer's data, followed by the rest. (This means the bytes in *data* are not reversed, they are simply prepended as a unit to the buffer' s original data). If a buffer write error occurs, subsequent writes do nothing.

```
int BufWriteError (BUF_TYPE b);
```

Returns non-zero if a write error occurred for the given buffer. Write errors occur if the buffer runs out of space for data or cannot allocate another data block (depends on the buffer type).

3.14.3 Buffer Configuration

The runtime library's encode and decode routines as well as the generated code access the buffers via the nine buffer macros described in the last two sections. These macros can be defined to call simple macros for speed or to call functions. Note that the buffer configuration is bound at the time the library and generated code are compiled.

The following is from `.../include/asn-config.h` and shows how to configure the buffer routines. This setup will make all calls to *BufGetByte* in the library and generated code call your *ExpBufGetByte* routine; the other buffer routines are mapped to their *ExpBuf* equivalents in a similar way.

```
#include "exp-buf.h"

#define BUF_TYPE ExpBuf **

#define BufGetByte( b) ExpBufGetByte (b)

#define BufGetSeg( b, lenPtr) ExpBufGetSeg (b, lenPtr)

#define BufCopy( dst, b, lenPtr) ExpBufCopy (dst, b, lenPtr)

#define BufSkip( b, len) ExpBufSkip (b, len)

#define BufPeekByte( b) ExpBufPeekByte (b)

#define BufPutByteRv( b, byte) ExpBufPutByteRv (b, byte)

#define BufPutSegRv( b, data, len) ExpBufPutSegRv (b, data, len)

#define BufReadError( b) ExpBufReadError (b)

#define BufWriteError( b) ExpBufWriteError (b)
```

If you want to use your own buffer type, simply edit the `asn-config.h` file such that it includes your buffer's header file, sets the *BUF_TYPE* type, and defines the nine buffer routines (*BufGetByte* etc.) to call your buffer routines. Your buffer routines should have the semantics and prototypes described in the last two sections (Sections 5.13.1 and 5.13.2). For eSNACC, only the SBuf is valid; other buffer types may work, but have not been tested on our releases.

3.14.4 SBuf Buffers

These buffers are still defined, but will not implement many of the newer features. The recommended/supported buffer type remains ExpBuf ONLY!

The *Sbufs* are simple buffers of a fixed size, much like an *ExpBuf* that cannot expand. If you attempt to write past the end of the buffer, the writeError flag will be set and the encoding will fail. If you attempt to read past the end of a buffer the readError flag will be set and the decoding will fail.

The *Sbufs* are useful if you can put a reasonable upper bound on the size of the encodings you will be dealing with. The buffer operations are much simpler because the data is contiguous. In fact, all of the *Sbuf* buffer operations are implemented by macros.

Look in `.../c-examples/simple/sbuf-ex.c` for a quick introduction to using *Sbuf*s in your code. The following operations are defined for the *Sbuf* buffers.

```
/* The nine required buffer operations */
#define SBufSkip( b, skipLen) ...
#define SBufCopy( dst, b, copyLen) ...
#define SBufPeekByte( b) ...
#define SBufGetSeg( b, lenPtr) ...
#define SBufPutSegRvs( b, seg, segLen) ...
#define SBufGetByte( b) ...
#define SBufPutByteRvs( b, byte) ...
#define SBufReadError( b) ...
#define SBufWriteError( b) ...
/* other useful buffer operations */
#define SBufInit( b, data, dataLen) ...
#define SBufResetInReadMode( b) ...
#define SBufResetInWriteRvsMode( b) ...
```

```
#define SBufInstallData( b, data, dataLen) ...
#define SBufDataLen( b) ...
#define SBufDataPtr( b) ...
#define SBufBlkLen( b) ...
#define SBufBlkPtr( b) ...
#define SBufEod( b) ...
```

eSNACC is configured to use *Sbuf*s by default. The symbols that will affect the buffer configuration during compilation of the libraries and generated code are *USE_EXP_BUF* and *USE_MIN_BUF*

3.15 Error Management

The decoding routines use *longjmp* to handle any errors they encounter in the value being decoded. *longjmp* works by rolling back the stack to where the *setjmp* call was made. Every decode routine takes a *jmp_buf* parameter (initialized by the *setjmp* call) that tells the *longjmp* routine how to restore the processor to the correct state. *longjmp* makes the error management much simpler since the decoding routines do not have to pass back error codes or check ones from other decoding routines.

Before a PDU can be decoded, the *jmp_buf env* parameter to the decoding routine must be initialized using the *setjmp* routine. This should be done immediately and only once before calling the decoding routine. This parameter will be passed down to any other decoding routines called within a decoding routine. The following code fragment from `.../c-examples/simple/exbuf-ex.c` shows how to use *setjmp* before decoding.

```
if (( val = setjmp (env)) == 0)
        BDecPersonnelRecord (& buf, &pr, &decodedLen, env);
else
{
        decodeErr = TRUE;
        fprintf (stderr, "ERROR - Decode routines returned %d\ n",
val); }
```

The code that will signal an error typically looks like:

```
if (mandatoryElmtCount1 != 2)
{
        Asn1Error (" BDecChildInformationContent: ERROR - non-
optional elmt missing”);
        longjmp (env, -108);
```

```
}
```

Most *longjmp* calls are preceded by a call to *Asn1Error* which takes a single *char ** string as a parameter. The library routines and the generated code try to use meaningful messages as the parameter *Asn1Error* is defined in `.../c-lib/inc/asn-config.h` and currently just prints the given string to *stderr*. You may wish to make it do nothing, which may shrink the size of your binary because all of the error strings will be gone. *Asn1Warning* is similar but is not used by the library or generated code anymore.

The encoding routines do no error checking except for buffer overflows. Hence, they do not use the *longjmp* mechanism and instead require you to check the status of the buffer after encoding (use *BufWriteError*()). If you are not building your values properly, for example having random pointers for uninitialized OPTIONAL elements, the encode routines will fail, possibly catastrophically.

4 C++ Code Generation

4.1 Introduction

The basic model that the generated C++ uses is similar to that of the generated C, but benefits from the object oriented features of C++. As with C, two files are generated for each ASN.1 module, a `.cpp` and a `.h` file (the original version created .C source files; this was modified to .cpp for portability between MS Windows and the various Unix platforms). If there are multiple .asn1 modules to compile, it is possible to build these .cpp/.h support files one at a time by using the eSNACC command line parameter “-I”. The “-I” directory reference must contain all of the .asn1 files containing the IMPORT references in order to get a successful compile (only the directory needs to be specified, very convenient). The eSNACC compiler will also take all .asn1 files on one command line input.

For error management C++ 's *try* and *throw* have replaced the *setjmp longjmp* used by the C decoders. This is the new SnaccException class. A “const char *” is available to any “catch” block of code to indicate the error (see an example test in ./SNACC/c++-examples/src/main.cpp).

C++ templates are very attractive for type safe lists (for SET OF and SEQUENCE OF) without duplicating code. Templates are used for all lists in the eSNACC C++ run-time library (in the original version, each list generates its own new class with all of the standard list routines).

As with the C code generation chapter, we will use the EX1 module to help illustrate some of the code generation. The following is the same EX1 module used in the C section.

```
EX1 DEFINITIONS::=
BEGIN
anOidVal OBJECT IDENTIFIER ::= {joint-sio-ccitt 40 foobar(29}
theSameOidVal OBJECT IDENTIFIER ::= (2 40 29}
anIntVal INTEGER ::= 1
aBoolVal BOOLEAN ::= TRUE
T1 ::= SEQUENCE
{
        INTEGER OPTIONAL,
        OCTET STRING OPTIONAL,
        ENUMERATED {a(0), b(1), c(2)},
```

```
        SEQUENCE OF INTEGER,
        SEQUENCE {id OBJECT IDENTIFIER, value OCTET STRING},
        CHOICE {INTEGER, OBJECT IDENTIFIER}
}
END
```

4.2 ASN.1 to C++ Naming Conventions

The C++ name for a type or value is the same as its ASN.1 name with any hyphens converted to underscores.

When an ASN.1 type or value name (after converting any hyphens to underscores) conflicts with a C++ keyword or the name of a type in another ASN.1 module (name clashes within the same ASN.1 scope are considered errors and are detected earlier), the resulting C++ class name will be the conflicting name with digits appended to it. A recent feature added to the ASN.1 syntax parsing will allow users to specify a specific C++ “namespace” for that module. This will prevent symbol conflicts between modules, if enabled.

Empty field names in SETs, SEQUENCEs, and CHOICEs will be filled. The field name is derived from the type name for that field. The library types such as INTEGER etc. have default field names defined by the compiler (see `../compiler/back-ends/c++-gen/rules.c`). The first letter of the field name is in lower case. Empty field names should be fixed properly by adding them to the ASN.1 source.

New type definitions will be generated for SETs, SEQUENCEs, CHOICEs, ENUMERATED, INTEGERs with named numbers and BIT STRINGs with named bits whose definitions are embedded in other SET, SEQUENCE, SET OF, SEQUENCE OF, or CHOICE definitions. The name of the new type is derived from the name of the type in which it was embedded and will be made unique by appending digits if necessary.

4.3 ASN.1 to C++ Class Translation

This section describes how C++ classes are used to represent each ASN.1 type. First, the general characteristics of each ASN.1 type's C++ class will be discussed followed by how the aggregate types (SETs, SEQUENCEs, CHOICEs, SET OFs, and SEQUENCE OFs) are represented. The representations of non-aggregate types (INTEGER, BOOLEAN, OCTET STRING, BIT STRING, OBJECT IDENTIFIER) and ANY and ANY DEFINED BY types are presented

in the next chapter since they form part of the C++ ASN.1 runtime library.

Every ASN.1 type is represented by a C++ class with the following characteristics:

1. it inherits from the *AsnType* base class
2. it has a default constructor (no parameters)
3. it has a copy constructor
4. it has a destructor
5. it has a clone method, *Clone*
6. it has an assignment operator
7. it has a content encode and decode method, *BEncContent* and *BDecContent*
8. it has a PDU encode and decode method, *BEnc* and *BDec*
9. it has a top level interfaces to the PDU encode and decode methods (handles the *exceptions,* etc.) for the user, *BEncPdu* and *BDecPdu*
10. it has a print method, *Print*, a virtual function that gets called from a global <<- operator
11. it has a PDU encode and decode method, PEnc and PDec (provided the

-b flag has been set)

If the metacode has been enabled (untested in recent releases):

11. it has a virtual function *_getdesc* that returns the classes meta description
12. if it is a structured type, it has a virtual function *_getref* that returns a pointer to one of its components/ members, specified through its name

If the Tcl code has been enabled (untested in recent releases):

13. it has a virtual function *TclGetDesc* to access the metacode's *_getdesc* routine from Tcl
14. it has a virtual function *TclGetVal* to retrieve an instance's value
15. it has a virtual function *TclSetVal* to change an instance's value
16. for SET, SEQUENCE, SET OF and SEQUENCE of: it has a virtual function *TclUnSetVal* to clear OPTIONAL members or to delete list elements, respectively

The following C++ fragment shows the class features listed above in greater detail.

```
Class Foo: public AsnType
{
        …// data members
public:
        Foo();
        Foo (const Foo &);
        Foo();
AsnType *Clone() const;
Foo     &operator = (const Foo &);
// content encode and decode routines
AsnLen  BEncContent(BUF_TYPE b);
void    BDecContent(BUF_TYPE b, AsnTag tag, AsnLen elmtLen,
             AsnLen &bytesDecoded, ENV_TYPE env);
// PDU (tags/lengths/content) encode and decode routines
AsnLen  BEnc (BUF_TYPE b);
void    BDec (BUF_TYPE b, AsnLen &bytesDecoded, ENV_TYPE env);

AsnLen  PEnc (BUF_TYPE b);
void    PDec (BUF_TYPE b, AsnLen &bitsDecoded);

// methods most likely to be used by your code
// Returns non-zero for success
int     BEncPdu (BUF_TYPE b, AsnLen &bytesEncoded);
int     BDecPdu (BUF_TYPE b, AsnLen &bytesDecoded);
void    Print (ostream &os) const;
#if META
        const AsnTypeDesc     *_getdesc() const;
        AsnType *getref (const char *membername, bool create= false);
#if TCL
        int       TclGetDesc(Tcl_DString *)const;
        int       TclGetVal(Tcl_Interp *)const;
        int       TclSetVal(Tcl_Interp *, const char *valstr);
        int       TclUnsetVal(Tcl_Interp *, const char *membername);
```

```
#endif
#endif
};
```

BEnc and *BDec* are PDU encode and decode methods. *BEnc* encodes the tag and length pairs for the object's type as well as the content (the object's value) to the given buffer, *b*, and returns the number of bytes written to the buffer for the encoding.

BDec decodes the expected tag and length pairs as well as the content of the object it is invoked upon from the given buffer, *b*, and increments *bytesDecoded* by the byte length of the tag(s), length(s) and value decoded. A SnaccException will be thrown in the case of an error.

BEncContent and *BDecContent* only deal with the content of the type their object represents. *BEncContent* encodes the object's value to the given buffer, *b*.

BDecContent decodes the object's value from the given buffer, *b*. The last tag and length pair on the content must be passed in via the *tag* and *elmtLen* parameters. The *tag* although always present, will only be used when decoding OCTET STRING and BIT STRING related types, to determine whether the encoding is constructed. The *elmtLen* is the length of the content and may be the indefinite length form. *bytesDecoded* is incremented by the actual number of bytes in the content; this is normally the same as *elmtLen* unless the indefinite length form was decoded. A SnaccException will be thrown if any decoding error occurs. The possible decoding errors depend on the type that is being decoded.

BEncPdu and *BDecPdu* are top-level interfaces to the PDU encode and decode routines. They present the simplest interface; they return TRUE if the operation succeeded and FALSE if an error occurred. *BEncPdu* checks for any buffer writing errors and *BDecPdu* checks for any buffer reading errors.

PEnc encodes data using the Packed Encoding Rules (PER). It NEVER encodes tags and sometimes does not encode a length determinant.

PDec decodes data that was encoded using the Packed Encoding Rules.

The *Print* method prints the object's value in ASN.1 value notation. When printing SETs and SEQUENCEs, a global variable is used for the current indent.

The *AsnType* base class, parameterless constructor and *Clone* method are required by the ANY and ANY DEFINED BY type handling mechanism explained in Sections 7.4 and 7.14. In brief,

the *AsnType* provides a base type that has virtual *BEnc*, *BDec* and *Clone* routines. The *Clone* routine is used to generate a new instance (not a copy) of the object that it is invoked on. This allows the ANY DEFINED BY type decoder to create a new object of the correct type from one stored in a hash table, when decoding (the *Clone* routine calls the parameterless constructor). The virtual *BEnc* and *BDec* are called from *AsnAny BEnc* and *BDec* methods.

The meta routines and the Tcl interface will be described in chapters 8 and 9, respectively.

4.3.1 Optional C++ namespace Designation

It is possible to specify a C++ namespace for any individual ASN.1 module through the use of the eSNACC “--snacc” pre-processor directive. For example:

--snacc namespace: "VDATestModule2Namespace"

following the ASN.1 module “BEGIN” statement will specify the namespace “VDATestModule2Namespace”. All associated ASN.1 references to any elements in this .asn1 file will be appropriately referenced.

4.3.2 SET and SEQUENCE

SET and SEQUENCE types generate classes that have their components as public data members. This makes accessing the components similar to referencing the fields of a C struct. For example the *T1* type in module EX1 will produce the following C++ class:

```
Class T1: public AsnType
{
public:
        AsnInt     *integer;
        AsnOcts    *octs;
        T1Enum     t1Enum;
        T1SeqOf    t1 SeqOf;
        T1Seq      t1 Seq;
        T1Choice   t1 Choice;
                   T1();
                   T1(const T1 &);
                   T1();
        AsnType    *Clone() const;
```

```
        T1          &operator=(const T1 &);
        AsnLen      BEnc(BUF_TYPE b);
        void        BDec(BUF_TYPE b, AsnLen &bytesDecoded, ENV_TYPE
                    env);
        AsnLen      BEncContent(BUF_TYPE b);
        void        BDecContent(BUF_TYPE b, AsnTag tag, AsnLen
elmtLen,
                    AsnLen &bytesDecoded, ENV_TYPE env);
        int  BEncPdu (BUF_TYPE b, AsnLen &bytesEncoded);
        int  BDecPdu (BUF_TYPE b, AsnLen &bytesDecoded);
        void Print (ostream &os) const;
#if META
        static const AsnSequenceTypeDesc        _desc;
        static const AsnSequenceMemberDesc      mdescs[];
        const AsnTypeDesc           *_getdesc() const;
        AsnType    *_getref (const char *membername, bool create =
false);
#if TCL
        int        TclGetDesc(Tcl_DString *)const;
        int        TclGetVal(Tcl_Interp *)const;
        int        TclSetVal(Tcl_Interp *, const char *valstr);
        int        TclUnsetVal(Tcl_Interp *, const char *membername);
#endif  // TCL
#endif // META
};
```

All OPTIONAL components in a SET or SEQUENCE are referenced by pointer. The constructor will automatically set OPTIONAL fields to *NULL*. The other methods are as described at the beginning of this section.

SETs and SEQUENCEs must contain all non-OPTIONAL components and SEQUENCEs must be ordered, otherwise an error is reported. Tagging errors are also reported. All detected errors cause a SnaccException to be thrown.

4.3.3 CHOICE

Each CHOICE type generates a class that has an anonymous union to hold the components of the CHOICE and a *choiceId* field to indicate which component is present.

Anonymous (un-named) unions allow you to reference the choice components with just the field name of the component; this makes referencing the contents of a CHOICE the same as referencing the contents of a SET or SEQUENCE.

The *choiceId* field contains a value in the *ChoiceIdEnum* that indicates the CHOICE field that is present. The names in the enumeration are derived from the field names of the CHOICE components.

When building a local value to be encoded, you must be sure to set the *choiceId* such that it corresponds to the value in the union. The decoder will set the *choiceId* when decoding incoming values.

Tagging errors cause a SnaccException to be thrown.

The following C++ class is produced for the CHOICE in the EX1 module.

```
Class T1Choice: public AsnType
{
public:
        enum ChoiceIdEnum
        {
             integerCid = 0;
             oidCid = 1;
        };
        enum ChoiceIdEnum choiceId
        union
        {
             AsnInt            *integer;
             AsnOid            *oid;
        };
                   T1Choice ();
                   T1Choice (const T1Choice &);
                   T1Choice ();
        AsnType    *Clone() const;
        T1Choice   &operator=(const T1Choice &);
        AsnLen     BEncContent(BUF_TYPE b);
        void       BDecContent(BUF_TYPE b, AsnTag tag, AsnLen
elmtLen,
                         AsnLen &bytesDecoded, ENV_TYPE env);
        AsnLen     BEnc(BUF_TYPE b);
```

```
        Void      BDec(BUF_TYPE b, AsnLen &bytesDecoded, ENV_TYPE
env);
        int       BEncPdu (BUF_TYPE b, AsnLen &bytesEncoded);
        int       BDecPdu (BUF_TYPE b, AsnLen &bytesDecoded);
        void      Print (ostream &os) const;
#if META
        static const Asn ChoiceTypeDesc         _desc;
        static const Asn ChoiceMemberDesc       mdescs[];
        const AsnTypeDesc     *_getdesc() const;
        AsnType   *_getref (const char *membername, bool create =
false);
#if TCL
        int             TclGetDesc(Tcl_DString *)const;
        int             TclGetVal(Tcl_Interp *)const;
        int             TclSetVal(Tcl_Interp *, const char *valstr);
#endif  // TCL
#endif // META
};
```

4.3.4 SET OF and SEQUENCE OF

Neither SET OF nor SEQUENCE OF types produce its own list class, the list template uses a single generic list type for all lists. This makes the C++ list routines type safe which allows the C++ compiler to detect more programmer errors.

Any tagging errors cause a SnaccException to be thrown. From the EX1 ASN.1 module the following list is produced:

```
class T1SeqOf : public AsnSeqOf<AsnInt>
{
public:
  virtual const char * typeName(void) const { return "T1SeqOf"; }
};
```

AsnSeqOf and AsnSetOf are both based off of the AsnList template. The AsnList template is taken directly from and inherets all of the functionality of the standard template C++ library std::list. An append() function was added to the AsnList template to manipulate the std::list::insert() function. This change was made in eSNACC version 1.7.

4.3.5 ENUMERATED, Named Numbers and Named Bits

The C++ type generator encapsulates each ENUMERATED type, INTEGER with named numbers and BIT STRING with named bits in a new class that inherits from the proper base class and defines the named elements. This provides a separate scope for these identifiers so their symbol will be exactly the same as their ASN.1 counterpart. Currently these identifiers are not checked for conflicts with C++ keywords, so you may have to modify some of them in the ASN.1 modules.

Inheritance is used for attaching ENUMERATED, named number and named bit information. ENUMERATED types inherit from the *AsnEnum* class, INTEGERs with named number types inherit from the *AsnInt* class and BIT STRINGs with named bits inherit from the *AsnBits* class.

If the tagging on the type is different from the type it inherits from, the PDU encode and decode methods are redefined with the correct tags to override the PDU encode and decode methods of the base class.

As with the other types, any tagging errors are reported and a SnaccException is thrown. No range checking is done on the decoded values although it would be easy to provide a new *BDecContent* method in the new class that calls the base class's and then checks the range of the result.

```
/* ENUMERATED { a (0), b (1), c (2) }  */
class  T1Enum: public AsnEnum
{
public:
  T1Enum(): AsnEnum() {}
  T1Enum (int i): AsnEnum (i) {}
  enum
  {
    a = 0,
    b = 1,
    c = 2
  };
};
```

4.4 ASN.1 to C++ Value Translation

C++ *const* values are used to hold ASN.1 defined values. C++ values will be produced for INTEGER, BOOLEAN and OBJECT IDENTIFIER ASN.1 values. An *extern* declaration for each *const* value is written at the end of the header file of the value's module. The *const* values are defined at the beginning of the `.cpp` file of the value's module. The *extern* declarations are at the end of the header file so that any required class definitions are available.

The following is from the end of the header file generated for the EX1 module:

```
Extern const AsnOid          anOidVal;
Extern const AsnOid          theSameOidVal;
Extern const AsnInt          anIntVal;
Extern const AsnBool         aBoolVal;
```

The following is from the beginning of the `.cpp` file generated for the EX1 module:

```
const AsnOid           anOidVal(“2.40.29”);
const AsnOid           theSameOidVal(“2.40.29”);
const AsnInt           anIntVal(1);
const AsnBool          aBoolVal(true);
```

The C++ constructor mechanism is used to generate these values. This mechanism is superior to C static initialization because it allows C++ code to be run to initialize the values.

4.5 Compiler Directives

Compiler directives are ignored by the C++ backend of eSNACC. If you want to implement them, look at the *FillCxxTypeDefInfo* routine in file `.../compiler/back-ends/c++-gen/types.c`. Then look at the way it is done for the C back-end (file `.../compiler/back-ends/c-gen/type-info.c`)

4.6 Compiling the Generated C++ Code

When compiling the generated C++ code on a Unix platform you will need:

1. The include directory where the file is from `.../c++-lib/inc/` have been installed in your include path so that the C++ sources can include these library header files. The header files should be included with statements like *#include*

<snacc/c++/asn-incl.h> and your C++ compiler should be supplied with `-I/usr/local/include` in case eSNACC got installed under `/usr/local/`. (For MS Windows, this would be ./SMPDist/include/esnacc/c++).

2. to link with the C++ ASN.1 runtime library, `.../c++-lib/libasn1c++.a`. In case eSNACC got installed under `/usr/local/`, your linker may need to be supplied with `-L/usr/local/lib` and `-lasn1c++` as arguments. (For MS Windows, this file is in ./SMPDist/lib/cppasn1[_d].lib; the .dll file is also stored in the %windir%/system32 directory).

3. to link with the math library (`-lm`), since the ASN.1 REAL type's encode and decode routine use some math routines.

For MS Windows, the include and library files are placed into ./SMPDist/include/esnacc and ./SMPDist/lib directories, where SMPDist is in the same directory as ./SNACC (this source). The library name is “cppasn1.lib”; the “cppasn1.dll” file is automatically placed into the %windir%/system32 directory.

See the example in `.../c++-examples/src/` for a complete example. The makefile and main routines are probably the most important. There are several other examples in the `.../c++-examples/`directory.

5 C++ ASN. 1 Library

5.1 Overview

The following sections describe the C++ representation of the non-aggregate ASN.1 types, ANY and ANY DEFINED BY types and the buffer and memory management. These classes and routines make up the C++ ASN.1 runtime library. Every aggregate ASN.1 type will be composed of these library types. The source files for this library are in `.../c++-lib/inc/` and `.../c++-lib/src/`.

As mentioned in the last chapter, each ASN.1 type is represented by a C++ class, which inherits from the *AsnType* base class. In addition to the standard encode, decode, print and clone methods described in the last chapter, each ASN.1 type class in the library may also have special constructors and other routines that simplify their use.

Unlike the classes generated for some of the aggregate types such as SETs and SEQUENCEs, the library types' data members are typically protected and accessed via methods.

All of the library classes' *BDec* routines will report tagging errors by throwing a SnaccException.

The top level PDU encode and decode methods are the same for all library types so they are defined as macros in `.../c++-lib/inc/asn-config.h`. For clarity's sake, the macro that is used to define these methods in the library type class definitions will be replaced with the actual prototypes.

5.2 Tags

The C++ tags are identical to those used in eSNACC's C ASN.1 environment. As with the C representation of tags, 4 byte long integers limit the maximum representable tag code to 2^{21}. Again, this should not be a problem.

No tags are encoded when PER is used.

5.3 Lengths

The C++ representation of lengths is the same at the C representation described earlier. Decoded lengths are represented by unsigned long integers, with the maximum value indicating indefinite length.

eSNACC users cannot set indefinite lengths when encoding constructed values' lengths. Of course, the generated decoders can handle both forms. Primitive values are always encoded with definite lengths as required by the standard; this is necessary to avoid confusion between a value's content and the End-Of-Contents marker.

The BDecLen routine will throw an exception if it attempts to read past the end of the data or the decoded length is too large to be held in the AsnLen representation. BDecEoc will report an error if it attempts to read past the end of the data or one of the EOC (End-Of- Contents) octets is non- zero.

Where PER is used, fragmentation applies. Any object that has a length greater than or equal to 64k elements will vary. PER only encodes definite lengths. It is possible to have a PER encoding with no length determinant.

5.4 The AsnType Base Class

Every ASN.1 type's C++ class uses the *AsnType* as its base class. The *AsnType* base class provides the following virtual functions:

- the destructor
- Clone()
- BEnc()
- BDec()
- PEnc()
- PDec()
- Print()
- _getdesc() (metacode)
- _getref() (metacode)
- TclGetDesc() (Tcl interface)
- TclGetVal() (Tcl interface)
- TclSetVal() (Tcl interface)
- TclUnsetVal() (Tcl interface)

The *AsnType* class is defined as follows:

```
class SNACCDLL_API AsnType
{
public:
```

```
  virtual         ~AsnType();

  virtual AsnType      *Clone() const=0;

  virtual void        BDec (const AsnBuf &b, AsnLen
&bytesDecoded)=0;
  virtual AsnLen      BEnc (AsnBuf &b) const =0 ;

  virtual void        PDec (const AsnBufBits &b, AsnLen
&bitsDecoded)=0;
  virtual AsnLen      PEnc (AsnBufBits &b) const =0 ;

  bool BEncPdu (AsnBuf &b, AsnLen &bytesEncoded) const;
  bool BDecPdu (const AsnBuf &b, AsnLen &bytesDecoded);

  virtual void        Print (std::ostream &) const=0;

  virtual const char * typeName(void) const { return "AsnType"; }

#if META
  static const AsnTypeDesc   _desc;

  virtual const AsnTypeDesc   *_getdesc() const;
  virtual AsnType *_getref (const char *membername, bool
create=false);

private:
  const char         *_typename() const;

#if TCL
public:
  virtual int         TclGetDesc (Tcl_DString *) const;
  virtual int         TclGetVal (Tcl_Interp *) const;
  virtual int         TclSetVal (Tcl_Interp *, const char *val);
  virtual int         TclUnsetVal (Tcl_Interp *, const char
*membernames);
#endif // TCL
#endif // META
```

```
};
```

The *AsnType* class and its virtual functions were added to support the ANY DEFINED BY type handling mechanism. This mechanism is described in this chapter.

Even if you do not use the ANY or ANY DEFINED BY types, the *AsnType* base class may be useful for adding features that are common to all of the types, such as changing the *new* and *delete* functions to improve performance.

Virtual functions provide the simplest method of handling ANY DEFINED BY and ANY types. Unfortunately, calls to virtual functions are slower than calls to normal functions due to their indirect nature. If you do not need support for the ANY DEFINED BY or ANY types you can remove most of the virtual functions to improve performance by undefining the *SUPPORT_ANY_TYPE* symbol (see the `asn-incl.h` file).

Note that a virtual destructor is included in the *AsnType* base class as well. This is done to make sure the *delete* routine always gets the correct size. See pages 215– 217 of Stroustrup [15] for a discussion of this.

5.5 BOOLEAN

The BOOLEAN type is represented by the *AsnBool* class. The following is the class definition of *AsnBool* from the.

```
class SNACCDLL_API AsnBool: public AsnType
{
protected:
  bool            value;
public:
   AsnBool (const bool val=false): value (val) {}
   virtual AsnType    *Clone() const {return new AsnBool(*this);}
   operator bool() const         { return value; }

   AsnBool &operator =(bool newvalue){ value = newvalue; return
*this; }
   AsnLen         BEnc (BUF_TYPE b);
   void           BDec (BUF_TYPE b, AsnLen &bytesDecoded);
   AsnLen         BEncContent (BUF_TYPE b);
```

```
   void           BDecContent (BUF_TYPE b, AsnTag tagId, AsnLen
elmtLen, AsnLen &bytesDecoded);


   AsnLen         BEnc (BUF_TYPE b);
   void           BDec (BUF_TYPE b, AsnLen &bitsDecoded);


   void           Print (std::ostream &) const;
   void           PrintXML (std::ostream &os, const char
*lpszTitle=NULL) const;


#if META
  static const AsnBoolTypeDesc   _desc;
  const AsnTypeDesc      *_getdesc() const;


#if TCL
  int            TclGetVal (Tcl_Interp *) const;
  int            TclSetVal (Tcl_Interp *, const char *val);
#endif // TCL
#endif // META
};
```

The *operator bool()* is defined such that when an *AsnBool* value is cast to a boolean, it returns the C++ style boolean value of the *AsnBools* value. There is also a constructor for *AsnBool* that builds an *AsnBool* value from the given C++ style boolean value. These two methods allow you to manipulate and access *AsnBool* values in a straight forward way as the following code illustrates.

```
Message::Send()
{
        AsnBool  okToSend;
        bool connectionOpen;
        bool  pduOk;
        …
        okToSend = connectionOpen && pduOk; // assign AsnBool from
bool
        if (okToSend) // cast AsnBool to bool
}
```

The *AsnBool* class contains the standard encode and decode methods that were described in the “C” library description.

BDecContent will throw a SnaccException if the length of an encoded BOOLEAN value's content is not exactly 1 octet.

Note that the *Clone* method returns an *AsnType ** value instead of an *AsnBool **. It might be more obvious to return an *AsnBool ** since due to single inheritance an *AsnBool* is also an *AsnType*. However, it must return an *AsnType ** for it to override the virtual function *Clone* defined in the *AsnType*

The *Print* method will print either “TRUE” or “FALSE” depending on the *AsnBool* value. No newline or other formatting characters are printed. The global indent information does not affect the output from this method.

5.6 INTEGER

The INTEGER type is represented by the *AsnInt* class. Improvements have been made to this class that allow for BIG Integers (greater than 4 bytes long). It is also possible to package these integer values based on the signed/unsigned ASN.1 rules for convenience when encoding/decoding big integers.

The following is the class definition of *AsnInt* from the `.../c++-lib/inc/asn-incl.h` file.

```
class SNACCDLL_API AsnInt : public AsnType
{
private:
   std::basic_string<unsigned char> bytes;
   void storeDERInteger(const unsigned char *pDataCopy, long
dataLen, bool unsignedFlag);
public:
   AsnInt (AsnIntType val=0);
   AsnInt (const char *str, bool unsignedFlag = true);
   AsnInt (const AsnOcts &o, bool unsignedFlag = true);
   AsnInt (const char *str, const size_t len, bool unsignedFlag =
true);
   AsnInt (const AsnInt &that);
   ~AsnInt ();

   virtual AsnType        *Clone() const {return new AsnInt(*this);}
   operator AsnIntType() const;
   bool        operator == (AsnIntType o) const;
```

```
   bool        operator != (AsnIntType o) const       { return !
operator==(o);}
   bool        operator == (const AsnInt &o) const;
   bool        operator != (const AsnInt &o) const;


   long length(void) const { return bytes.length();}
   const unsigned char * c_str(void) const { return bytes.c_str(); }
   void getPadded(unsigned char *&data, size_t &len, const size_t
padToSize=0) const;


   void        Set(const unsigned char *str, size_t len, bool
unsignedFlag=true);
   void        Set(AsnIntType i);


   AsnLen         BEnc (BUF_TYPE b);
   void           BDec (BUF_TYPE b, AsnLen &bytesDecoded);
   AsnLen         BEncContent (BUF_TYPE b);
   void           BDecContent (BUF_TYPE b, AsnTag tagId, AsnLen
elmtLen, AsnLen &bytesDecoded);


   AsnLen         PEnc (BUF_TYPE b);
   void           PDec (BUF_TYPE b, AsnLen &bitsDecoded);



   virtual const char * typeName(void) const { return "INTEGER"; }
   void      Print (std::ostream &os) const;
   void      PrintXML (std::ostream &os, const char *lpszTitle=NULL)
const;


#if META
  static const AsnIntTypeDesc   _desc;
  const AsnTypeDesc      *_getdesc() const;
#if TCL
  int            TclGetVal (Tcl_Interp *) const;
  int            TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

5.7 ENUMERATED

The ENUMERATED type is represented by the *AsnEnum* class. The following is the class definition of *AsnEnum.*.

```
class SNACCDLL_API AsnEnum: public AsnInt
{
public:
#if !TCL
            AsnEnum(): AsnInt() {}
#endif
            AsnEnum (int i): AsnInt (i) {}


            virtual AsnType  *Clone() const{return new
AsnEnum(*this);}


  AsnLen         BEnc (BUF_TYPE b);
  void           BDec (BUF_TYPE b, AsnLen &bytesDecoded);

  AsnLen         PEnc (BUF_TYPE b);
  void           PDec (BUF_TYPE b, AsnLen &bitsDecoded);


#if META
  static const AsnEnumTypeDesc   _desc;
  const AsnTypeDesc      *_getdesc() const;
#if TCL
  int            TclGetVal (Tcl_Interp *) const;
  int            TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

Note that it is not derived from *class AsnType* directly but from *AsnInt* instead.

5.8 NULL

The NULL type is provided by the *AsnNull* class. This class has no data members and includes only the standard methods.

```
class SNACCDLL_API AsnNull: public AsnType
{
public:
  virtual AsnType     *Clone() const {return new AsnNull(*this);}

  AsnLen         BEncContent (BUF_TYPE /*b*/)      { return 0; }
  void           BDecContent (BUF_TYPE b, AsnTag tagId, AsnLen
elmtLen, AsnLen &bytesDecoded);
  AsnLen         BEnc (BUF_TYPE b);
  void           BDec (BUF_TYPE b, AsnLen &bytesDecoded);

  AsnLen         PEnc (BUF_TYPE b);
  void           PDec (BUF_TYPE b, AsnLen &bitsDecoded);

  void           Print (std::ostream &os) const;
  void           PrintXML (std::ostream &os, const char
*lpszTitle=NULL) const;

#if META
  static const AsnNullTypeDesc   _desc;
  const AsnTypeDesc      *_getdesc() const;
#if TCL
  int            TclGetVal (Tcl_Interp *) const;
  int            TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

5.9 REAL

REAL types are represented by the *AsnReal* class. Internally, a *double* is used to hold the real value. The following is from `.../c++-lib/inc/asn-incl.h`:

```
class SNACCDLL_API AsnReal: public AsnType
{
protected:
```

```
  double         value;
public:
   AsnReal():value (0.0){}
   AsnReal (double val):value (val){}


   virtual AsnType *Clone() const { return new AsnReal(*this);}
                  operator double() const         { return value; }
   AsnReal       &operator = (double newvalue)    { value =
newvalue; return *this; }


   AsnLen         BEncContent (BUF_TYPE b);
   void           BDecContent (BUF_TYPE b, AsnTag tagId, AsnLen
elmtLen, AsnLen &bytesDecoded);
   AsnLen         BEnc (BUF_TYPE b);
   void           BDec (BUF_TYPE b, AsnLen &bytesDecoded);


   AsnLen         PEnc (BUF_TYPE b);
   void           PDec (BUF_TYPE b, AsnLen &bitsDecoded);


   void           Print (std::ostream &os) const;
   void           PrintXML (std::ostream &os, const char
*lpszTitle=NULL) const;


#if META
  static const AsnRealTypeDesc   _desc;
  const AsnTypeDesc      *_getdesc() const;
#if TCL
  int            TclGetVal (Tcl_Interp *) const;
  int            TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */
};
```

The *double* representation and support routines can depend on the compiler or system you are using so several different encoding routines are provided. Even so, you may need to modify the code.

There are three content encoding routines included and they can be selected by defining one of *IEEE_REAL_FMT* or

IEEE_REAL_LIB, or nothing. Defining *IEEE_REAL_FMT* uses the encode routine that assumes the double representation is the standard IEEE double [3]. Defining *IEEE_REAL_LIB* uses the encode routine that assumes the IEEE functions library (isinf, scalbn, signbit etc.) is available. If neither are defined, the default encode routine uses *frexp*. Currently, the `.../configure` script has not got any checks for the IEEE format or library and therefore does not define any of the symbols. (This should be fixed.)

AsnReal constants are used to hold *PLUS_INFINITY* and *MINUS_INFINITY* values. These values are initialized using the *AsnReal* constructor mechanism with the *AsnPlusInfinity* and *AsnMinusInfinity* routines. If you do not define *IEEE_REAL_FMT* or *IEEE_REAL_LIB,* you should rewrite the *AsnPlusInfinity* routine such that it is correct for your system.

There is only one content decoding routine and it builds the value through multiplication and the *pow* routine (requires the math library). The content decoding routine only supports the binary encoding of a REAL, not the decimal encoding.

5.10 BIT STRING

The BIT STRING type is represented by the *AsnBits* class.

```
class SNACCDLL_API AsnBits: public AsnType
{
  AsnBits(const char *stringForm=NULL);
  AsnBits (size_t numBits)
          { bits=NULL;nblFlag = false; Set (numBits); }
  AsnBits (const char *bitOcts, size_t numBits)
          {bits=NULL; nblFlag = false; Set (bitOcts, numBits); }
  AsnBits (const AsnBits &b)      { bits=NULL; Set (b); }
  ~AsnBits();


  virtual AsnType         *Clone() const {return new
AsnBits(*this);}


  AsnBits         &operator = (const AsnBits &b)      { Set (b);
return *this; }


  // overwrite existing bits and bitLen values
  void            Set (size_t numBits);
```

```
  void            Set (const char *bitOcts, size_t numBits);
  void            Set (const AsnBits &b);


  AsnBits &       operator = (const char *stringForm);
  bool            operator == (const AsnBits &ab) const   { return
BitsEquiv (ab); }
  bool            operator != (const AsnBits &ab) const   { return !
BitsEquiv (ab); }


  bool            soloBitCheck(size_t);
  void            SetBit (size_t);
  void            ClrBit (size_t);
  bool            GetBit (size_t) const;
  bool            IsEmpty() const;


  void            UseNamedBitListRules(bool flag) { nblFlag =
flag; }


  size_t         BitLen() const            { return bitLen; }
  const char *   data() const { return bits; }
  size_t         length() const { return ((bitLen+7)/8); }


  AsnLen         BEncContent (AsnBuf &b) const;
  void           BDecContent (const AsnBuf &b, AsnTag tagId, AsnLen
elmtLen, AsnLen &bytesDecoded);
  AsnLen         BEnc (AsnBuf &b) const;
  void           BDec (const AsnBuf &b, AsnLen &bytesDecoded);


  AsnLen         PEnc (BUF_TYPE b);
  void           PDec (BUF_TYPE b, AsnLen &bitsDecoded);



  void            Print (std::ostream &) const;
  void            PrintXML (std::ostream &os, const char
*lpszTitle=NULL) const;


#if META
  static const AsnBitsTypeDesc   _desc;
```

```
  const AsnTypeDesc      *_getdesc() const;

#if TCL
  int            TclGetVal (Tcl_Interp *) const;
  int            TclSetVal (Tcl_Interp *, const char *val);
#endif /* TCL */
#endif /* META */

private:
  bool            BitsEquiv (const AsnBits &ab) const;
  void            BDecConsBits (const AsnBuf &b, AsnLen elmtLen,
AsnLen &bytesDecoded);

protected:
  size_t       bitLen;
  char         *bits;
  bool         nblFlag;
};
```

The *AsnBits* class contains a pointer to the bits and an integer that holds the length in bits of the BIT STRING.

In addition to the standard methods, the *AsnBits* class has methods for initializing and comparing bit string values and methods for setting and getting individual bits in a value.

An *AsnBits* value can be created three ways: from the number of bits, from a *char *** and its bit length or from another *AsnBits* value. Look at the constructors and the *set* and *reset* methods.

SetBit and *ClrBit* can be used for setting the values of individual bits in the BIT STRING value. Given the bit's index, *SetBits* sets that bit to one. *ClrBit* sets the bit of the given index to zero. The bit indexes start at zero, with zero being the first (most significant) bit in the BIT STRING. *GetBit* will return *true* if the specified bit is one and *false* if the bit is zero. If the given bit index is too large, *SetBit* and *ClrBit* do nothing and *GetBit* returns *false*.

The *==* and *!=* operators have been overloaded such that given two *AsnBits* values, they will behave as expected.

Each *AsnBits* value stores its bit string in a single contiguous block of memory. Received BIT STRING values that were encoded in the

constructed form are converted to the simple, flat form (see Section 5.8). eSNACC provides no facility for encoding or internally representing constructed BIT STRING values.

5.11 OCTET STRING

OCTET STRING values are represented with the *AsnOcts* class.

```
class SNACCDLL_API AsnOcts: public AsnType

{

public:

   // constructor always copies strings so destructor can always
delete

   AsnOcts() {m_pFileSeg = NULL;}

   AsnOcts (const char *str) { m_pFileSeg = NULL; m_str.assign(str);
}

   AsnOcts (const char *str, const size_t len) { m_pFileSeg = NULL;
m_str.assign(str, len); }

   AsnOcts (const AsnOcts &o);

//   AsnOcts (const std::filebuf &fb);

   virtual ~AsnOcts();


   virtual AsnType     *Clone() const { return new AsnOcts(*this);}


   AsnOcts      &operator = (const AsnOcts &o)  { m_str = o.m_str;
return *this; }

   AsnOcts      &operator = (const char *str)   { m_str.assign(str);
return *this; }


   void         Set (const char *str, size_t len);


   size_t       Len() const;


   const std::string &   data() const;

   const char *          c_str() const;

   const unsigned char * c_ustr() const;


   bool operator == (const AsnOcts &o) const;

   bool operator != (const AsnOcts &o) const;
```

```
   AsnLen         BEncContent (AsnBuf &b) const;
   void           BDecContent (const AsnBuf &b, AsnTag tagId, AsnLen
elmtLen, AsnLen &bytesDecoded);
   AsnLen         BEnc (AsnBuf &b) const ;
   void           BDec (const AsnBuf &b, AsnLen &bytesDecoded);


   AsnLen         PEnc (BUF_TYPE b);
   void           PDec (BUF_TYPE b, AsnLen &bitsDecoded);


   void           Print (std::ostream &os) const;
   void           PrintXML (std::ostream &os, const char
*lpszTitle=NULL,
                      const char *lpszType=NULL) const;


private:
  void           BDecConsOcts (const AsnBuf &b, AsnLen elmtLen,
AsnLen &bytesDecoded);
  void           FillStringDeck(AsnBuf &b, AsnLen elmtLen, AsnLen
&bytesDecoded);


  // IF AsnOcts length is < MAX_OCTS, store in AsnString base
  mutable std::string  m_str;


  // IF AsnOcts length is > MAX_OCTS, store in m_FileBuf;
  mutable AsnFileSeg     *m_pFileSeg;
};
```

The *AsnOcts* class contains a pointer to the octets and an integer that holds the length in octets of the OCTET STRING.

There are four constructors for *AsnOcts*. The parameterless constructor will initialize the octet string to zero length with a *NULL* octets pointer. The constructor that takes a single *char ** assumes that the given string is NULL terminated and initializes the octet pointer with a pointer to a copy of the given string and sets the *octetLen* to the *strlen* of the string (this does not usually include the NULL terminator). The constructor that takes *char ** and a length, *len,* initializes the octets pointer to point to a copy of *len* characters from the given string and sets the *octetLen* to *len*. The last constructor will initialize an *AsnOcts* value by copying the given *AsnOcts* value.

As with the BIT STRING content decoder, OCTET STRING content decoder can handle constructed values. These are handled in the same way as the constructed BIT STRING values; they are converted to the simple contiguous representation. Every OCTET STRING value will automatically have a NULL terminator appended to it; this extra character will not be included in the string's length and will make some strings easier to deal with for printing etc.

The *operator char *()* is defined for the *AsnOcts* class to return a pointer to the octets. The *Len* method returns the length in bytes of the string value. These may be useful for passing the octets to other functions such as *memcpy* etc.

The *==* and *!=* operators have been overloaded such that given two *AsnOcts* values, they will behave as expected.

AsnOcts now takes advantage of the multiple-buffer “AsnFileSeg” class, which allows a limited stream capability. This allows very large Octet String values to be stored efficiently, not copied multiple times when decoding.

5.12 Built-in Strings PrintableString, BMPString, TeletexString, NumericString, VideotexString, T61String, IA5String, GraphicString, VisibleString, ISO646String, GeneralString, UniversalString, UTF8String, UTCTime, GeneralizedTime

The eSNACC run-time library directly supports the various ASN.1 string definitions. This includes validating the actual data restrictions for the various ASN.1 string types (i.e. encode operations will fail if invalid characters are being encoded). All strings support some basic load/unload operations. Some of the strings will handle wide characters (e.g. BMPString, UTF8String). The UTCTime and GeneralizedTime strings are built-in due to the custom ASN.1 tag in the encoding; no format checking is performed on these strings. All of the string classes as well as some support classes are presented below. Here are some notes on translating strings between the various formats.

For any wide character load operation from an 8 bit extended character, the following example will work to load the value into a multi-byte ASN.1 string, like UTF8String.

```
UTF8String  utf8String;
std::string AAAB(strWithExtendedCharPresent);
```

```
        std::wstring AAAC;
        for (unsigned int ii=0; ii < AAAB.length(); ii++)
            AAAC += AAAB[ii];   // ASSIGN each 1byte char to
wchar.
        UTF8String SNACC_utf8String(AAAC);


// DEFINE some extra functionality to the "*String" classes using
similar names
class SNACCDLL_API AsnString : public std::string, public AsnType
{
public:
       AsnString(const char* str = NULL) { operator=(str); }
       AsnString(const std::string& str) { operator=(str); }


       AsnString& operator=(const char* str);
       AsnString& operator=(const std::string& str)  { assign(str);
return *this; }


//     const std::string & get();
//     void set(const AsnString &o)


       int cvt_LDAPtoStr  (char *in_string, char  **char_ptr);
       int cvt_StrtoLDAP  (wchar_t *in_string, char  **char_ptr);


       // returns a null terminated 'C' string
   //  const char *c_str() const;


       AsnLen BEnc(AsnBuf &b) const;
       void   BDec(const AsnBuf &b, AsnLen& bytesDecoded);
       AsnLen BEncContent(AsnBuf &b) const;
       void   BDecContent(const AsnBuf &b, AsnTag tagId, AsnLen
       elmtLen, AsnLen& bytesDecoded);


       // each string type must implement these methods
       //
       // tagCode should be implemented to return the ASN.1 tag
       // corresponding to the character string type.
```

```
        //
        // the check method should be implemented to enforce any
rules imposed on the
        // character string type.
        virtual BER_UNIV_CODE tagCode(void) const = 0;
        virtual bool check() const{ return true; }

        void Print(std::ostream& os) const;
        void PrintXML(std::ostream& os, const char *lpszTitle) const;

private:
        void BDecConsString(const AsnBuf &b, AsnLen elmtLen, AsnLen
&bytesDecoded);
};



class SNACCDLL_API NumericString : public AsnString
{
public:
        NumericString(const char* str = NULL) : AsnString(str)
        {}
        NumericString(const std::string& str) : AsnString(str)
        {}

        NumericString& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        NumericString& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }

    AsnType* Clone() const{ return new NumericString(*this); }

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return NUMERICSTRING_TAG_CODE;
}
        bool check() const;                     // Enforce string
rules
};
```

```
class SNACCDLL_API PrintableString : public AsnString
{
public:
        PrintableString(const char* str = NULL) : AsnString(str)  {}
        PrintableString(const std::string& str) : AsnString(str)  {}

        PrintableString& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        PrintableString& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }

    AsnType* Clone() const{ return new PrintableString(*this); }

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return
PRINTABLESTRING_TAG_CODE; }
        bool check() const;                     // Enforce string
rules
};



class SNACCDLL_API TeletexString : public AsnString
{
public:
        TeletexString(const char* str = NULL) : AsnString(str)
        {}
        TeletexString(const std::string& str) : AsnString(str)
        {}

        TeletexString& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        TeletexString& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }

    AsnType* Clone() const{ return new TeletexString(*this); }

        // ASN.1 tag for the character string
```

```
        BER_UNIV_CODE tagCode() const{ return TELETEXSTRING_TAG_CODE;
}
};

// T61String -- Alternate name for TeletexString
typedef TeletexString T61String;



class SNACCDLL_API VideotexString : public AsnString
{
public:
        VideotexString(const char* str = NULL) : AsnString(str)
        {}
        VideotexString(const std::string& str) : AsnString(str)
        {}

        VideotexString& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        VideotexString& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }

    AsnType* Clone() const{ return new VideotexString(*this); }

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return
VIDEOTEXSTRING_TAG_CODE; }
};

class SNACCDLL_API IA5String : public AsnString
{
public:
        IA5String(const char* str = NULL) : AsnString(str)        {}
        IA5String(const std::string& str) : AsnString(str)        {}

        IA5String& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        IA5String& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }
```

```
    AsnType* Clone() const{ return new IA5String(*this); }

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return IA5STRING_TAG_CODE; }
        bool check() const;                     // Enforce string
rules
};


class SNACCDLL_API GraphicString : public AsnString
{
public:
        GraphicString(const char* str = NULL) : AsnString(str)
        {}
        GraphicString(const std::string& str) : AsnString(str)
        {}

        GraphicString& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        GraphicString& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }

    AsnType* Clone() const{ return new GraphicString(*this); }

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return GRAPHICSTRING_TAG_CODE;
}
};


class SNACCDLL_API VisibleString : public AsnString
{
public:
        VisibleString(const char* str = NULL) : AsnString(str)
        {}
        VisibleString(const std::string& str) : AsnString(str)
        {}
```

```
        VisibleString& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        VisibleString& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }

    AsnType* Clone() const{ return new VisibleString(*this); }

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return VISIBLESTRING_TAG_CODE;
}
        bool check() const;                     // Enforce string
rules
};

// ISO646String -- Alternate name for VisibleString
typedef VisibleString ISO646String;



class SNACCDLL_API GeneralString : public AsnString
{
public:
        GeneralString(const char* str = NULL) : AsnString(str)
        {}
        GeneralString(const std::string& str) : AsnString(str)
        {}

        GeneralString& operator=(const char* str)
             { AsnString::operator=(str); return *this;}
        GeneralString& operator=(const std::string& str)
             { AsnString::operator=(str); return *this; }

    AsnType* Clone() const{ return new GeneralString(*this); }

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return GENERALSTRING_TAG_CODE;
}
};
```

```
// Multi-byte character based definitions... BMP, Universal, UTF8.
class SNACCDLL_API WideAsnString : public std::wstring, public
AsnType
{
public:
        WideAsnString(const char* str = NULL)
        { set(str); }
        WideAsnString(const std::string& str)
        { set(str.c_str()); }
        WideAsnString(const std::wstring& wstr)
        { assign(wstr); }

        // each string type must implement these methods
        //
        // tagCode should be implemented to return the ASN.1 tag
corresponding to the
        // character string type.
        //
        // the check method should be implemented to enforce any
rules imposed on the
        // character string type.
        virtual BER_UNIV_CODE tagCode() const = 0;
//      virtual bool check() const = 0;

        AsnLen BEnc(AsnBuf &b) const;
        void   BDec(const AsnBuf &b, AsnLen &bytesDecoded);
        virtual AsnLen BEncContent(AsnBuf &b) const = 0;
        virtual void   BDecContent(const AsnBuf &b, AsnTag tagId,
AsnLen elmtLen,
             AsnLen &bytesDecoded) = 0;

        void Print(std::ostream &os) const;
        void PrintXML(std::ostream &os, const char *lpszTitle) const;

        void set(const char* str);
        void getAsUTF8(std::string &utf8String) const;
        char* getAsUTF8() const;
```

```
protected:
        AsnLen CombineConsString(const AsnBuf &b, AsnLen elmtLen,
std::string& encStr);
};

class SNACCDLL_API BMPString : public WideAsnString
{
public:
        BMPString(const char* str = NULL) : WideAsnString(str)
            {}
        BMPString(const std::string& str) : WideAsnString(str)
            {}
        BMPString(const std::wstring& wstr) : WideAsnString(wstr)
        {}

        BMPString& operator=(const char* str)
            { set(str); return *this;}
        BMPString& operator=(const std::string& str)
            { set(str.c_str()); return *this; }
        BMPString& operator=(const std::wstring& wstr)
            { assign(wstr); return *this; }

        const char* typeName() { return "BMPString"; }
        AsnType* Clone() const { return new BMPString(*this); }

        AsnLen BEncContent(AsnBuf &b) const;
        void   BDecContent(const AsnBuf &b, AsnTag tagId, AsnLen
elmtLen,
            AsnLen &bytesDecoded);

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return BMPSTRING_TAG_CODE; }
//      bool check() const;               // Enforce string rules
};


class SNACCDLL_API UniversalString : public WideAsnString
{
```

```
public:
        UniversalString(const char* str = NULL) : WideAsnString(str)
             {}
        UniversalString(const std::string& str) : WideAsnString(str)
             {}
        UniversalString(const std::wstring& wstr) :
WideAsnString(wstr)     {}


        UniversalString& operator=(const char* str)
             { set(str); return *this;}
        UniversalString& operator=(const std::string& str)
             { set(str.c_str()); return *this; }
        UniversalString& operator=(const std::wstring& wstr)
             { assign(wstr); return *this; }


        const char* typeName(){ return "UniversalString"; }
        AsnType* Clone() const{ return new UniversalString(*this); }


        AsnLen BEncContent(AsnBuf &b) const;
        void   BDecContent(const AsnBuf &b, AsnTag tagId, AsnLen
elmtLen,
             AsnLen &bytesDecoded);


        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const           { return
UNIVERSALSTRING_TAG_CODE; }
//      bool check() const;               // Enforce string rules
};



class SNACCDLL_API UTF8String : public WideAsnString
{
public:
        UTF8String(const char* str = NULL) : WideAsnString(str)
        {}
        UTF8String(const std::string& str) : WideAsnString(str)
        {}
        UTF8String(const std::wstring& wstr) : WideAsnString(wstr)
             {}
```

```
        UTF8String& operator=(const char* str)
             { set(str); return *this;}
        UTF8String& operator=(const std::string& str)
             { set(str.c_str()); return *this; }
        UTF8String& operator=(const std::wstring& wstr)
             { assign(wstr); return *this; }

        const char* typeName(){ return "UTF8String"; }
        AsnType* Clone() const{ return new UTF8String(*this); }

        AsnLen BEncContent(AsnBuf &b) const;
        void   BDecContent(const AsnBuf &b, AsnTag tagId, AsnLen
elmtLen,
             AsnLen &bytesDecoded);

        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const{ return UTF8STRING_TAG_CODE; }
//      bool check() const;               // Enforce string rules
};



// Time Classes
//
class SNACCDLL_API UTCTime : public VisibleString
{
public:
        UTCTime(const char* str = NULL) : VisibleString(str){}
        UTCTime(const std::string& str) : VisibleString(str){}
        virtual ~UTCTime()
                  {}

        UTCTime& operator=(const char* str)
             { VisibleString::operator=(str); return *this;}
        UTCTime& operator=(const std::string& str)
             { VisibleString::operator=(str); return *this; }
```

```
    AsnType* Clone() const                              { return new
UTCTime(*this); }


        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const                   { return
UTCTIME_TAG_CODE; }
};



class SNACCDLL_API GeneralizedTime : public VisibleString
{
public:
        GeneralizedTime(const char* str = NULL) : VisibleString(str)
        {}
        GeneralizedTime(const std::string& str) : VisibleString(str)
        {}
        virtual ~GeneralizedTime(){}

        GeneralizedTime& operator=(const char* str)
             { VisibleString::operator=(str); return *this;}
        GeneralizedTime& operator=(const std::string& str)
             { VisibleString::operator=(str); return *this; }

    AsnType* Clone() const{ return new GeneralizedTime(*this); }


        // ASN.1 tag for the character string
        BER_UNIV_CODE tagCode() const                   { return
GENERALIZEDTIME_TAG_CODE; }
};
```

5.13 OBJECT IDENTIFIER

OBJECT IDENTIFIER values are represented with the *AsnOid* class.

```
class SNACCDLL_API AsnOid : public AsnRelativeOid
{
public:
        AsnOid();
        AsnOid(const char* pszOID);
```

```
        AsnOid(const AsnOid &that):AsnRelativeOid(that) {m_isRelative
= false;}


        virtual AsnType* Clone() const                      { return
new AsnOid(*this); }
        virtual const char* typeName() const          { return
"AsnOid"; }


        // get a copy of the OID's NULL-terminated dotted string
notation
        char* GetChar() const       { return strdup(operator const
char*()); }


        // set from a number-dot null term string
        void PutChar(const char* szOidCopy)
        { Set(szOidCopy); }


        AsnOid operator+(const AsnRelativeOid& ro) const;
        AsnOid& operator+=(const AsnRelativeOid& ro);


#if META
        static const AsnOidTypeDesc _desc;
        const AsnTypeDesc* _getdesc() const;
#endif /* META */
};
```

The *AsnOid* stores OBJECT IDENTIFIER values in their encoded form to improve performance. It seems that the most common operation with OBJECT IDENTIFIERs is to compare for equality, for which the encoded representation (which is canonical) works well.

The *AsnOid* is very similar to the *AsnRelativeOid* class in all respects, except that its content is required to have a minimum of two arc numbers.

class SNACCDLL_API AsnRelativeOid : public AsnType

{

public:

```
        AsnRelativeOid()
        { Init(); }
        AsnRelativeOid(const AsnRelativeOid& o)             { Init();
Set(o); }
        AsnRelativeOid(const char* pszOID)                  { Init();
Set(pszOID); }
        virtual ~AsnRelativeOid();


        AsnRelativeOid& operator=(const AsnRelativeOid& o)
        { Set(o); return *this; }
        operator const char*() const;


        virtual AsnType* Clone() const                      { return new
AsnRelativeOid(*this); }
        virtual const char* typeName() const    { return
"AsnRelativeOid"; }
        size_t Len() const                                  { return
octetLen; }
        const char* Str() const                             { return
oid; }


        bool operator==(const AsnRelativeOid& o) const    { return
OidEquiv(o); }
        bool operator==(const char* o) const;
        bool operator!=(const AsnRelativeOid& o) const    { return
!operator==(o); }
        bool operator!=(const char* o) const              { return
!operator==(o); }
        bool operator<(const AsnRelativeOid& o) const;


        unsigned long NumArcs() const;
        void GetOidArray(unsigned long oidArray[]) const;


        void Set(const char* szOidCopy);
        void Set(const char* encOid, size_t len);
        void Set(const AsnRelativeOid& o);
```

```
        void Set(unsigned long arcNumArr[], unsigned long
arrLength);

        AsnLen BEnc(AsnBuf& b) const;
        void   BDec(const AsnBuf& b, AsnLen& bytesDecoded);
        AsnLen BEncContent(AsnBuf& b) const;
        void   BDecContent(const AsnBuf& b, AsnTag tagId, AsnLen
elmtLen,
               AsnLen& bytesDecoded);

        AsnLen PEnc(AsnBufBits& b) const;
        void   PDec(AsnBufBits& b, AsnLen& bitsDecoded);

        void   Print(std::ostream& os, unsigned short indent = 0)
const;
        void   PrintXML(std::ostream& os, const char* lpszTitle =
NULL) const;

#if META
        static const AsnRelativeOidTypeDesc _desc;
        const AsnTypeDesc* _getdesc() const;

#if TCL
        int TclGetVal(Tcl_Interp* ) const;
        int TclSetVal(Tcl_Interp* , const char* val);
#endif /* TCL */
#endif /* META */

protected:
        size_t          octetLen;
        char*                   oid;
        mutable char*   m_lpszOidString;
        bool            m_isRelative;
```

```
private:
        void Init() { octetLen = 0; oid = NULL; m_lpszOidString =
NULL;
             m_isRelative = true; }
        bool OidEquiv(const AsnRelativeOid& o) const;
        void createDottedOidStr() const;
};
```

The *AsnRelativeOid* class has four constructors, which are similar to those of the *AsnOcts* class. A special constructor that takes arc numbers as parameters and uses default parameters is provided. Unlike a RELATIVE OID, an OBJECT IDENTIFIER value must have at least two arc numbers so the first two parameters do not have default values. All of the other parameters are optional; since their default value of *-1* is an invalid arc number (they must be positive) they will not be used in the value. For example to build the value *{1 2 3}* you simply use *AsnOid ("1.2.3").* This constructor is convenient but is more expensive in terms of CPU time than the others.

The *operator char *()* is defined for the *AsnRelativeOid* class to return a pointer to the encoded (RELATIVE) OBJECT IDENTIFIER value. The *Len* method returns the length in bytes of the encode (RELATIVE) OBJECT IDENTIFIER value (NOT the number arcs in the value). These may be useful for passing the octets to other functions such as *memcpy* etc. *NumArcs* returns the number of arcs that the value is comprised of.

The *==* and *!=* operators have been overloaded such that given two *AsnOcts* values, they will behave as expected.

5.14 SET OF and SEQUENCE OF

In the C ASN.1 library, the list type was in the library because it was generic and every SET OF and SEQUENCE OF was defined as an *AsnList*. In C++, a new class is defined every list from a C++ template, providing a type safe list mechanism. The AsnList class is based off of the standard template library std::list. These classes use the std::list template.

5.15 ANY and ANY DEFINED BY

The ANY DEFINED BY type can be handled automatically by eSNACC provided you use the SNMP OBJECT-TYPE macro to specify the identifier to type mappings. The identifier can be an INTEGER or OBJECT IDENTIFIER. Handling ANY types properly will require modifications to the generated code since there is no identifier associated with the type. If your definition is not handled by the SNMP OBJECT-TYPE macro(s) or the OID is not recognized when decoding, then the data is placed in an AsnBuf. These ANY buffers will perform encode/decode operations as normal eSNACC classes for consistency. The user has access to the data as a buffer.

Look at the C and C++ ANY examples and the `any.asn1` file included with this release for information on using the OBJECT-TYPE macro. Note that the OBJECT-TYPE macro has been modified slightly to allow INTEGER values (identifiers).

An ANY DEFINED BY type is represented by the *AsnAny* class.

```
/* this is put into the hash table with the int or oid as the key */
class SNACCDLL_API AnyInfo
{
public:
  int            anyId;   // will be a value from the AnyId enum
  AsnOid         oid;   // will be zero len/null if intId is valid
  AsnIntType     intId;
  AsnType        *typeToClone;
};


class SNACCDLL_API AsnAny: public AsnType
{
public:
   static Table         *oidHashTbl;   // all AsnAny class
SNACCDLL_API instances
   static Table         *intHashTbl;   // share these tables
   mutable AnyInfo      *ai;      // points to entry in hash tbl for
this type
   AsnType              *value;
   AsnBuf               *anyBuf; // used if ai == null
```

```
   AsnAny()            { ai = NULL; value = NULL; anyBuf=NULL;}
   AsnAny(const AsnAny &o) { ai = NULL; value = NULL; anyBuf = NULL;
*this = o; }
   virtual ~AsnAny();


   AsnAny &operator = (const AsnAny &o);
   virtual AsnType * Clone() const { return new AsnAny(*this); }


   static void AsnAnyDestroyHashTbls();


   // Recursive call to destroy table during destruction
   static void AsnAnyDestroyHashTbl(Table *&pHashTbl);


   // class SNACCDLL_API level methods
   static void         InstallAnyByInt (AsnIntType intId, int anyId,
AsnType *type);
   static void         InstallAnyByOid (AsnOid &oid,  int anyId,
AsnType *type);


   int            GetId()   const            { return ai ? ai->anyId
: -1; }
   void           SetTypeByInt (const AsnInt& id) const;
   void           SetTypeByOid (const AsnOid &id) const;


   AsnLen         BEnc (AsnBuf &b) const;
   void           BDec (const AsnBuf &b, AsnLen &bytesDecoded);


   void   Print (std::ostream &) const;
   void   PrintXML (std::ostream &os, const char *lpszTitle=NULL)
const;


#if TCL
   int            TclGetDesc (Tcl_DString *) const;
   int            TclGetVal (Tcl_DString *) const;
   int            TclSetVal (Tcl_Interp *, const char *val);
   int            TclUnSetVal (Tcl_Interp *, const char *member);
#endif /* TCL */
```

```
};
```

The C++ mechanism is similar to the C mechanism which uses hash tables to hold the identifier to type mappings. In this section we will discuss the main differences of the C++ ANY DEFINED BY handling mechanism. You should read Section 5.12 for caveats and other important information.

In C, the hash table entry held the size of the type and pointers to its encode, decode, free etc. routines to describe the type. In C++ these have been replaced with a pointer to an instance of the type. A hash table entry contains:

- The *anyId*
- the INTEGER or OBJECT IDENTIFIER that maps to it
- a pointer to an instance of the identified type

All C++ ASN.1 types use the *AsnType* base class, which designates the following functions as virtual:

- the destructor
- Clone()
- BDec()
- BEnc()
- Print()
- _getdesc() (metacode)
- _getref() (metacode)
- TclGetDesc() (Tcl interface)
- TclGetVal() (Tcl interface)
- TclSetVal() (Tcl interface)
- TclUnsetVal() (Tcl interface)

This allows the ANY DEFINED BY handling routines to treat a value of any ASN.1 type as an *AsnType*. So, for each type the ANY DEFINED BY handling code has access to the virtual methods. Note that the *value* field in the *AsnAny* class and the *typeToClone* field in the *AsnAny* class are both *AsnType* *

Before an ANY DEFINED BY value can be decoded, the field that contains its identifier must have been decoded and used with the *AsnAny* value's *SetTypeByInt* or *SetTypeByOid* methods. Then the ANY DEFINED BY value can be decoded by calling its (*AsnAny*)

BDec routine. This in turn calls the *Clone* routine on the type in the hash table entry to generate the correct object. Then the *BDec* method of the newly created object is called.

When the C ANY DEFINED BY decoder allocates a value, it uses the size information for the identified type. This is not safe for C++ so the virtual *Clone* routine was added to the *AsnType* base class. This allows the proper constructor mechanism to be used when allocating the value.

The virtual *Clone* routine simply calls its type's parameterless constructor via *new* (hence every ASN.1 type's class must have a parameterless constructor). *Clone* is a poor name since the routine only produces a new instance of the given type without copying the original's data.

The hash tables are automatically initialized using the C++ constructor mechanism. You do not need to call any initialization routines as described in the C chapter.

5.16 Buffer Management

The C++ buffer management provided with eSNACC has been re-written to accommodate lists of references to original data buffers/files. The following is from `.../c++-lib/inc/asn-buf.h`:

```
class SNACCDLL_API AsnBuf
{
public:
   AsnBuf();
   AsnBuf(const char *seg, size_t segLen);
   AsnBuf(const std::stringstream &ss);
   AsnBuf(std::streambuf *sb);
   AsnBuf(const AsnBuf &o);
   AsnBuf(const char *pFilename);
   ~AsnBuf()  { clear(); }

   void          PutByteRvs (char byte);
   void          PutSegRvs  (const char *seg, size_t segLen);
   void          PutStream(std::streambuf *sb);

   unsigned char GetUByte() const { return (unsigned char)
GetByte();}
   char          GetByte() const;
```

```
   unsigned long GetSeg(char *seg, long segLen) const;
   char *        GetSeg(long segLen) const;
   void          GetSeg(std::string &str, long segLen=0) const;
   AsnFileSeg *  GetFileSeg(long segLen) const;
   void          PutFileSeg(AsnFileSeg *fs);
   void          GrabAny(AsnBuf &anyBuf, SNACC::AsnLen
&bytesDecoded) const;
   const Deck &  deck() const {return m_deck;}
   const Card &  card() const {return **m_card;}
   unsigned long length() const;
   long          splice(AsnBuf &b);
   void          skip(size_t skipBytes);
   char          PeekByte() const;
   AsnBuf &      operator=(const AsnBuf &o);
   bool          operator==(const AsnBuf &b) const;
   bool          operator<(const AsnBuf &rhs) const;


   void          hexDump(std::ostream &os) const;


#ifdef _DEBUG
   void status(std::ostream &os);
#endif


   AsnBufLoc GetReadLoc() const;
   void      SetReadLoc(const AsnBufLoc &bl) const;


//#ifdef WIN32
//   void ResetMode(std::ios_base::open_mode mode =
std::ios_base::in) const;
//#else
   void ResetMode(std::ios_base::openmode mode = std::ios_base::in)
const;
//#endif


private:
   void clear();


protected:
```

```
   mutable SNACC::Deck::iterator m_card;
   mutable SNACC::Deck m_deck;
};
```

This is the only buffer type provided with the C++ library. It uses a sophisticated list of stream buffers to allow a buffer to be assigned as it is decoded, not copied, thus saving valuable system resources on large messages (i.e. on certain elements the decoded data elements contain references to the original input data buffer, not a copy of the data). This buffer scheme is also used with Octet String(s).

5.17 Error Management - SnaccException

The C++ ASN.1 error management is no longer identical to that of the C ASN.1 model. C++ exception handling (*try* and *throw*) are now being implemented by a new eSNACC class, SnaccException. The following is the class definition of *SnaccException* from the `.../c++-lib/inc/snaccexcept.h` file.

```
class SNACCDLL_API SnaccException: public std::exception
{
public:
   SnaccException() throw();
   SnaccException(const char *file, long line_number, const char
*function=NULL,
    const char *whatStr=NULL, long errorCode=DEFAULT_ERROR_CODE)
throw();
   virtual const char * what() const throw();
   void push(const char *file, long line_number, const char
*function=NULL) throw();
   virtual ~SnaccException() throw();


   void getCallStack(std::ostream &os);
   const CallStack * getCallStack(void) const { return stack; }
   SnaccException & operator=(const SnaccException &o);
   long  errorCode;


private:
    // first element in array is populated by the constructor.  All
    // other elements in the array are populated by push().
   short stackPos;
```

```
   CallStack stack[STACK_DEPTH];


protected:
   const char *whatStr;
};
```

SnaccException is an enhancement to the C++ library to use a standard Exception class derived from the C++ std::exception class that will enable applications to catch all exceptions using the standard class. It is important that the application specifically trap the SnaccException in order to see the details that caused the error.

C++ Test Sources and Code Demonstration

The demonstration program provided with the eSNACC distribution demonstrates .asn1 file compilation, references to these ASN.1/class items, and some demonstrations of loading and referencing data in eSNACC classes. Thread processing, exception processing, buffer handling, string tests, etc. are also demonstrated. These various features are shown in the ./SNACC/c++-examples/src directory ("C" features are demonstrated in the various subdirectories under ./SNACC/c-examples). Some of the test features are described below.

The various .asn1 files for this project must be compiled using the eSNACC compiler:

```
cd ../
..\..\..\SMPDist\bin\esnacc.exe -D -C  -I . rfc1157-snmp.asn1
```

Notice the "-I ." parameter; this allows the compiler to resolve the rfc1157-snmp.asn1 IMPORT references for the compile to succeed and produce the rfc1157-snmp.cpp and rfc1157-snmp.h source files. All 4 .asn1 files in this directory are compiled similarly. The "vdatest_asn1.asn1" source file demonstrates how to define an OID to syntax specification for an "ANY DEFINED BY" definition.

The single main program runs all of the various tests. The "snmp.cpp" source file runs the snmp V1 test suite for ASN.1 compliance; this test suite runs over 20,000 tests. Some are errors to show graceful handling of ASN.1 decode errors.

For the C++ library, the SNACC/c++-examples/src/vdatest.cpp source file provides examples of using all eSNACC C++ classes. This include ANY load/unload operations throught the ANY DEFINED BY feature as well as general ANY definitions (no OID to specify a data type). For the "C" library, the source file is SNACC/c-examples/vdatestC/vdatestC.c.

Bibliography

[1] CCITT. *Data Communications Networks Open systems Interconnection (OSI) Model and Notation, Service Definition* , chapter Recommendation X. 208, Specification of Abstract Syntax Notation One (ASN.1), pages 57–130. Number Fascicle VIII.4 in Blue Book. Omnicom, 115 Park St., S. E., Vienna, VA 22180 USA, November 1989.

[2] CCITT. *Data Communications Networks Open systems Interconnection (OSI) Model and Notation, Service Definition* , chapter Recommendation X. 209, Spec- ification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), pages 130– 151. Number Fascicle VIII. 4 in Blue Book. Omnicom, 115 Park St., S. E., Vienna, VA 22180 USA, November 1989.

[3] Motorola Inc. *MC68881 Floating-Point Coprocessor User's Manual* . Motorola Inc., 1985.

[4] ISO. Information technology—open systems interconnection—abstract syntax notation one (asn.1).

[5] ISO. Working paper for draft proposed international standard for information systems—programming language c++, 28 April 1995.

[6] Gerald Neufeld and Son Vuong. An overview of asn.1. *IEEE Networks and ISDN Systems* , 23(5): 393– 415, Feb 1992.

[7] Gerald Neufeld and Yeuli Yang. An asn.1 to c compiler. *IEEE Transactions on Software Engineering* , 16(10): 1209–1220, Oct 1990.

[8] OMG. The common object request broker: Architecture and specification. Technical report, OMG, 1993.

[9] John K. Ousterhout. *Tcl and the TK Toolkit* . Addison-Wesley Publishing Company, 1994. ISBN 0-201-63337-X.

[10] M. Rose and K. McCloghrie. Structure and identification of management information for tcp/ip-based internets (rfc 1155). Network Information Center, SRI International, May 1990.

[11] Marshall T. Rose. *ISODE, The ISO Development Environment: User Manual* . Wollongong Group, 1129 San Antonio Rd. Palo Alto, California, USA, February 1990.

[12] Michael Sample. How fast can asn.1 encoding rules go? Master's thesis, University of British Columbia, Vancouver, B. C. Canada V6T 1Z2, April 1993.

[13] Michael Sample and Gerald Neufeld. Implementing efficient encoders and decoders for network data representations. *IEEE INFOCOM '93 Proceedings* , 3: 1144–1153, Mar 1993.

[14] Douglas Steedman. *ASN.1, The Tutorial and Reference* . Technology Appraisals Ltd., 1990. ISBN 1 871802 06 7.

[15] Bjarne Stroustrup. *The C++ Programming Language, 2nd Edition*. Addison-Wesley Publishing Co., 1991. ISBN 0201539926.